OPIC
Office de la propriété intellectuelle du Canada

CIPO
Canadian Intellectual Property Office

(12)(19)(CA) **Demande-Application**

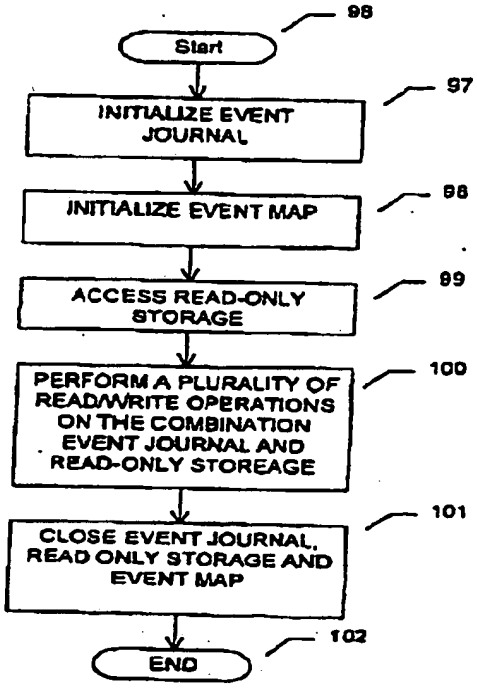
(21)(A1) **2,221,216**
(22) 1997/11/14
(43) 1998/05/15

(72) SQUIBB, Mark, US
(71) SQUIBB, Mark, US

(51) Int.Cl.[6] G06F 17/30
(30) 1996/11/15 (60/030,998) US

(54) **SYSTEME ET APPAREIL SERVANT A FUSIONNER UN JOURNAL D'EVENEMENTS D'ECRITURE ET LE CONTENU INITIAL D'UNE MEMOIRE POUR PRODUIRE UN CONTENU A JOUR A L'AIDE D'UN TABLEAU D'EVENEMENTS**

(54) **SYSTEM AND APPARATUS FOR MERGING A WRITE EVENT JOURNAL AND AN ORIGINAL STORAGE TO PRODUCE AN UPDATED STORAGE USING AN EVENT MAP**



(57) L'invention est constituée par une méthode et un appareil servant à restaurer une mémoire d'ordinateur à jour à partir d'un journal d'événements d'écriture, et dans lesquels la duplication du contenu initial de la mémoire engendre un tableau des événements établi à partir du journal des événements d'écriture. Ce tableau

(57) A method and apparatus for restoring an updated computer storage from a journal of write events and copy of an original storage generates an event map from the journal of write events. The event map permits efficient combination of the contents of the write event journal and the original storage. The event map also enables

OPIC
OFFICE DE LA PROPRIÉTÉ INTELLECTUELLE DU CANADA

CIPO
CANADIAN INTELLECTUAL PROPERTY OFFICE

(21) (A1) 2,221,216
(22) 1997/11/14
(43) 1998/05/15

permet de combiner efficacement le contenu du journal des événements d'écriture et le contenu initial de la mémoire. Il permet également de traduire le journal en une forme qui exprime la différence entre le contenu initial de la mémoire et le contenu mis à jour, ainsi que de fusionner de façon efficace un journal d'événements d'écriture et une bande en continu.

translation of the event journal into a delta expressing the differences between the original and updated storages. The event map similarly permits efficient merging of a write event journal and an original file stored streaming tape.

BNSDOCID: <CA_____2221216A1_I_>

Abstract of Disclosure:

A method and apparatus for restoring an updated computer storage from a journal of write events and a copy of an original storage generates an event map from the journal of write events. The event map permits efficient combination of the contents of the write event journal and the original storage. The event map also enables translation of the event journal into a delta expressing the differences between the original and updated storages. The event map similarly permits efficient merging of a write event journal and an original file stored streaming tape.

What Is Claimed Is:

1. A method for creating an event map from an event journal, comprising the steps of:

reading each of a plurality of write event entries from an event journal;

generating a current event marker for each of the plurality of write event entries; and

determining if an overlap condition exists between a latest current event marker and an existing current event marker and if an overlap condition exists, removing the overlap condition to represent the latest current event marker.

2. The method according to claim 1, wherein the step of determining and removing the overlap condition includes comparing the latest current event marker and the existing current event marker to identify an overlapped portion.

3. The method according to claim 1, wherein the event journal includes the plurality of write events stored in a computer memory device, the computer memory device having an array of storage units each having a predetermined address.

4. The method according to claim 3, wherein each of the plurality of write events includes at least an event address and an event data, the event address representing a location of the write event in the computer memory device and the event data including a content of the storage units occupied by the write event.

5. The method according to claim 1, wherein the current event marker includes at least a marker origin address, a marker event span and a marker data pointer, the marker origin address representing a location in the computer memory device, the marker event span representing a number of occupied storage units, and the marker data pointer representing a pointer to an event data in the computer memory device.

6. The method according to claim 1, further comprising the step of storing each of the plurality of current event markers in a sorted list.

7. The method according to claim 2, wherein the step of removing the overlap condition includes one of deleting the overlapped portion and revising the overlapped portion.

8. The method according to claim 3, wherein the array of storage units stores one of bytes in a computer file, blocks in a computer disk and records in a database.

9. The method according to claim 1, wherein the event journal is stored on a backup computer system, the backup computer system being coupled to a primary computer system, and further comprising the step of storing each current event marker on the backup computer system, and wherein the steps of generating the current event marker and determining and removing the overlap condition are performed on the backup computer system.

10. The method according to claim 1, wherein the steps of generating the current event marker and determining if an overlap condition exists are performed immediately after the step of reading each of the plurality of write event entries.

11. The method according to claim 5, further comprising the step of, when a write event entry occurs only at a block boundary and an event size equals a block size, storing each current event marker in an array of storage marker pointers containing one of a null value and the marker data pointer, the null value indicating an original storage block not changed by a write event entry and the marker data pointer indicating the original data block changed by the write event entry.

12. The method according to claim 1, further comprising the step of dividing the event journal into a plurality of segments and wherein the step of reading each of the plurality of write events includes reading each of the plurality of write events for a

respective one of the plurality of segments.

13. A method of fulfilling a read request for an updated storage using an original storage, an event journal and an event map, the method comprising the steps of:

receiving a read request, the read request including a data position and a read size;

identifying, from the read request and via the event map, portions of the read request to be provided by the event journal and portions of the read request to be provided by the original storage; and

fulfilling the read request.

14. The method according to claim 13, wherein the data position represents an offset from an origin of the updated storage and the read size represents a number of storage units to be read from the updated storage, the updated storage containing a data content of the original storage and subsequent changes to the original storage via a plurality of write events.

15. The method according to claim 13, wherein the original storage includes a computer memory device having an array of storage units each having a predetermined address, the event journal being stored in the computer memory device, the event journal including a plurality of write event entries.

16. The method according to claim 15, wherein each of the plurality of write event entries includes at least an event address and an event data, the event address representing a location of the write event in the computer memory device and the event data including a content of the storage units occupied by the write event.

17. The method according to claim 16, wherein the event map is created from the

event journal via:

reading each of the plurality of write event entries from the event journal;

generating a current event marker for each of the plurality of write event entries; and

determining if an overlap condition exists between a latest current event marker and an existing current event marker and if an overlap condition exists, removing the overlap condition to represent the latest current event marker.

18. The method according to claim 17, wherein the current event marker includes at least a marker origin address, a marker event span and a marker data pointer, the marker origin address representing a location in the computer memory device, the marker event span representing a number of occupied storage units, and the marker data pointer representing a pointer to an event data in the computer memory device.

19. The method according to claim 13, wherein the original storage is empty.

20. The method according to claim 13, wherein the original storage includes a read only storage device.

21. The method according to claim 13, wherein the step of fulfilling the read request includes reading the requested data from a respective one of the original storage and the event map as determined via the identifying step.

22. The method according to claim 21, wherein the step of fulfilling the read request further includes reading each contiguous segment of the original storage and recording a result of the read request on a second storage media.

23. The method according to claim 22, wherein the second storage media includes one of a streaming backup tape and a seekable disk.

24. The method according to claim 13, wherein the original storage resides on a streaming tape and the streaming tape is not repositioned during the step of fulfilling the read request except to skip overlaid data segments from the event journal.

25. The method according to claim 22, wherein the event journal includes a plurality of event journals.

26. A method for converting an event journal into a delta, comprising the steps of:

generating an event map from an event journal;

identifying, via the event map, a mismatching segment between an original storage and an updated storage;

recording a mismatch marker for each mismatching segment;

identifying, via the event map, a matching segment between the original storage and the updated storage; and

recording a match marker for each matching segment.

27. The method according to claim 26, wherein each match marker represents a position and a size of a matching segment in the original storage and the updated storage and each mismatch marker represents a position, a size and a data content of a mismatching segment between the original storage and the updated storage.

28. The method according to claim 26, wherein the step of identifying a mismatching segment includes identifying the mismatching segment as a function of an entry in the event map using a marker data pointer to supply a data content from

the event journal.

29. The method according to claim 26, wherein the step of identifying a matching segment includes identifying an omission from the event map and wherein the step of recording a match marker includes recording the match marker for each omission in the event map.

30. The method according to claim 26, wherein the updated storage is stored on a first storage device and wherein a copy of the original storage and the recorded markers are stored on a second storage device, the second storage device providing a backup of the updated storage.

31. The method according to claim 26, wherein the mismatch marker represents a size and a data content of a mismatching segment between the original storage and the updated storage.

32. The method according to claim 26, wherein the mismatch marker represents a position, a size and a marker data pointer of a mismatching segment between the original storage and the updated storage.

33. The method according to claim 30, wherein the first storage device is located on a first computer system and the second storage device is located on a second computer system.

34. The method according to 26, wherein the event journal includes a plurality of event journals, and wherein the generating, identifying and recording steps are performed for each of the plurality of event journals.

35. The method according to claim 26, wherein the event journal is recorded on a client server and the markers are recorded on a backup server.

36. The method according to claim 26, further comprising the step of dividing the event journal into a plurality of segments and wherein the steps of generating, identifying and recording are performed for each of the plurality of segments.

37. The method according to claim 1, wherein the event journal includes synchronization information and further comprising the step of identifying a termination condition as a function of the synchronization information.

38. The method according to claim 1, wherein the event journal includes a plurality of event journals and the steps of reading, generating and determining are performed for each of the plurality of event journals.

[56104501/15]

SYSTEM AND APPARATUS FOR MERGING A WRITE EVENT JOURNAL AND AN ORIGINAL STORAGE TO PRODUCE AN UPDATED STORAGE USING AN EVENT MAP

Reference to Paper Appendix

This application incorporates by reference the computer program listing contained in the attached paper appendix. The paper appendix includes 109 pages.

Field of the Invention

The present invention relates to improvements in the field of computer systems having backup/restore or archive/retrieve subsystems. More particularly, the present invention relates to a method and apparatus to efficiently protect and archive active data to streaming media.

Background Information

In a data processing system, a backup/restore subsystem, usually referred to as a backup subsystem, is typically used to save a recent copy of an active file and several earlier versions. There are, for example, three general strategies employed by backup systems. First, full backup periodically copies all files from a client system's storage to a backup server. A second strategy includes incremental backup, where the client system copies only the modified files to the backup server. In a third strategy, a delta backup copies only the modified portions of the modified files to the backup server.

Complete discussions of various backup and storage technologies are disclosed, for example in U.S. Patent No. 5,479,654 entitled APPARATUS AND METHOD FOR RECONSTRUCTING A FILE FROM A DIFFERENCE SIGNATURE AND AN ORIGINAL FILE, which is hereby incorporated by reference. Applicants co-pending applications entitled COMPUTER APPARATUS AND METHOD FOR MERGING A SEQUENTIAL PLURALITY OF DELTA STREAMS, Attorney Docket No. HP-10951196-1, filed _____, and COMPUTER APPARATUS AND METHOD

FOR MERGING SYSTEM DELTAS, Attorney Docket No. HP-10960109, filed November 30, 1995, also relate to backup and storage technologies and are hereby incorporated by reference. In addition, the Storage Service Kit, ©1996 by Mark Squibb also relates to data storage systems and is hereby incorporated by reference.

It is apparent to those skilled in the art that in any given backup system, the higher the backup frequency, the more accurate the backup copy will represent the present state of the data within a file. Considering the large volume of data maintained and continuously generated in a data processing system, the amount of storage, time and other resources associated with protecting data are very substantial. Thus, those skilled in the art are continuously engaged in searching for better and more efficient ways to provide data protection.

The time lag between the last backup, for example by the backup methods described above, and the current data on an active system represents risk of data loss. This "protection gap" is an active concern among computer users because it represents unprotected information. Mirroring systems, described below, partially overcome this gap.

It is well known in the art to capture write events to a storage system. For example, each time a change is made to a storage device, the change is recorded or logged into a second media. A variety of types of media have been used for recording of logs including, for example, streaming tape, hard disk, and remote hard disk.

A write event comprises, for example, a storage indicator indicating what storage component or device the write applies to, a position indicator within the component or event as an offset telling where in the storage the write occurred, and the data (e.g., event data) which was written. Various embodiments also include time or sequence markers for synchronization to identify or select points in time and to coordinate state information across storage boundaries. A collection of write events is known as an event log, referred to as an event journal when the event log is stored on a storage device.

Since events are recorded just after they occur, event logs are ordered in chronological sequence. Events at the beginning of the journal occurred before events at the end of the journal. Creation of event logs is well known in the art. Replaying event logs to re-enact changes to a storage component is also well known in the art.

Prior art systems use event journals to replay changes to random access writeable media. The word "replay" means, for example, to chronologically re-enact the storage write events that resulted in a particular file given an original file. The replay process begins with a disk file on random access storage and an event log synchronized with the disk file. The initial file's Data State, for example, must correspond to the starting instant of the event log. The events in the event log are repeated on the disk file in the sequence that they occur in the log.

Each event is read from the event journal in sequence from beginning to end. After each event is read, the corresponding event offset is located in the disk file. This location process usually involves repositioning in the disk file to a random position that may be before or after the position of the last event offset. The event data is then written to the file at the new offset. Old information in the file is destroyed because the new data overlays the prior data. This process is repeated until the event log is exhausted. When the process completes, data in the revised file represents the final Data State represented by the event log.

Mirror systems duplicate changes as they occur. Storage devices are usually treated as block devices. When change occurs, a write event is packaged and transmitted to a remote mirror system. Upon receipt, the remote mirror duplicates the change in the mirror storage. Mirror systems sometimes employ event logs to store events.

Event logs are used in mirroring systems for several purposes. For example, event logs are used in mirroring systems to compensate for transmission delays at the source system. At the mirror system, event logs are used to cache events when the mirrored storage cannot keep up with incoming write events. It is also known to temporarily halt incoming events to a mirror system so that the data to the mirror system is constant during backup of the mirror.

From a data protection perspective, there are two types of events that compromise data access. The first and best understood is a hardware failure. Mirroring effectively prevents a hardware failure from compromising business continuance. The second event type is a logical failure. A logical failure occurs when a user, operator or application does something that destroys, corrupts or distorts information. Mirroring systems immediately and irreversibly duplicate logical errors.

Mirroring systems have several deficiencies. For example, mirroring does not protect from logical failures. In addition, the need to re-execute every write event on another random-access storage device shortly after the initial event effectively requires duplication of all active storage in a second storage system. The cost of the extra second storage and the management overhead prevents widespread adoption of mirroring technology.

Mirroring does eliminate the data protection gap inherent with backup technology. Mirrored systems immediately protect new data, unless it is destroyed by a logical failure. The volatile nature of mirrored storage, however, is a deficiency. The cost of duplicating massive storage is also a deficiency. In contrast, the present invention provides the same data protection as mirrored storage at much lower cost and provides recourse for logical errors.

Clustering is a type of mirroring. Clusters are a collection of servers that maintain a mirror distributed among several other servers on the network. Clustered servers share the logical fault intolerance and storage doubling characteristics of mirrored systems.

File mirroring systems are another type of mirroring. It is known in the art to have a many-to-one mirror, for example when a single system provides a logical mirroring service for a number of network servers. The single system collects change from the servers on a network. The change is stored or applied to a hard disk cache of active data files. Periodically the data file versions are backed up to tape. File mirror systems, however, share the same deficiencies as mirror systems because all active data must be stored on disk and included archive service provides random archive

granularity. Also as indicated above, mirroring systems maintain a duplicate copy of the storage in case the primary storage system fails.

U.S. Patent No. 5,086,502 to Malcolm describes an apparatus and method for recording an event journal to a backup storage means to provide backup to a primary storage drive. The Malcolm patent, however, explicitly requires that a random access storage device hold the base file. It is important to note that this process occurs in the order of the events recorded in the event journal. These events are recorded in the order that changes were made to the base file, and therefore are random with respect to position in the base stream. If the Malcolm system fails, a sequential list of write operations is replayed on a copy of the original data to restore the data set to the latest working instance. It is well known in the art to store write events and combine them with a backup copy to recover a database as of the latest instant. Thus, this technique requires first installing the base copy on random access media and second repeating all write events to the base copy on the disk. Accordingly, data recovery using Malcolm journals is restricted to randomly seekable and writeable storage means.

For example, a first event may indicate a write event of 100 bytes at offset 1000 in a base file (e.g., an original file) and, a second event may indicate a write event of 100 bytes at an offset of 500 in the same base file. In order for the journal of Malcolm to be used to recover a file by the specified method: the base file must first be placed on random access media; a seek to byte 1000 must occur followed by a write of the 100 bytes from the first write event; and the primary media must than seek to byte 500 and the 100 bytes relating to the second write event must be written to the primary media. This requirement to seek in the base file multiple times effectively prevents streaming media from being used in combination with Malcolm's journals or in the common precedent of using replaying database redo logs.

It is also well known in the art to use streaming media to backup data files. For example, prior art systems operate by copying data files to a streaming media. Streaming media is preferred primarily because of its low cost. Backup systems tend to be used for infrequent retrieval, and when such retrieval is required, data is usually

required in the order in which it was recorded.

The combination of, for example, Malcolm and prior art backup systems does not contemplate operating in the absence of a primary storage media for the file to be restored or any means of combining a file stored on one streaming media to be merged with an event journal and written to another streaming storage. The prior art systems all require, for example, an intermediate step of placing the original file (which may be stored on streaming media) into a seekable media, such as a disk. Once the original file is on disk, then a history of write events can be written onto the original file, via seeking to the appropriate addresses on the disk, to recreate the latest version of the file.

In the field of information storage, a variety of media types are used. Streaming media, or tape, is dramatically cheaper than random access media. For most practical purposes, however, tape is not considered a readily seekable media. While most tape devices support positioning of media to a linear address, this positioning requires linear traversal of a very long media. This positioning takes a lot of time, and is used sparingly in practical applications.

Random access media permits information to be efficiently retrieved in an order different than it was laid out on the media. Streaming media is preferred for high volume applications, however, because of low cost and high capacity. Streaming media is much cheaper than equivalent random access media. Also, streaming tape devices have many times the capacity of random access devices. For example, a tape library may hold a thousand tapes, each tape having the capacity of 40 or more hard disks.

As a result, streaming storage devices are preferred places to store immense volumes of information. In this example, a single tape library is capable of holding as much information as 40,000 disk drives. The ability to concentrate and efficiently store huge volumes of information is a significant advantage in many applications, particularly when providing data protection services for large networks. The

combination of lower storage cost plus much higher capacity are extremely important factors with data protection systems.

It is therefore an object of the present invention to provide improved data protection including both backup and archive capability in a data processing environment.

It is a further object of the present invention to provide data protection including backup and archive services in a client/server environment.

It is a further object of the present invention to provide data protection by transferring a minimum amount of data across communication link.

It is a further object of the present invention to eliminate the data protection gap inherent to backup technology by protecting information up to the last instant using low-cost streaming media.

It is a further object of the present invention to protect data from software and user errors by providing a storage archive for older versions.

It is a further object of the present invention to use inexpensive streaming media, e.g., tape, for backup storage.

It is a further object of the present invention to provide a cost and time-effective method for providing an archive mirror using inexpensive streaming media.

It is a further object of the present invention to convert a write event journal into a delta.

It is a further object of the present invention to convert a write event journal into a map of changed segments that can be queried with respect to linear offset.

It is a further object of the present invention to enable use of a read-only base stream and an event log as a readable, seekable and writeable stream.

It is a further object of the present invention to provide an apparatus and method for combining a plurality of write event journals with a read-only non-seekable base stream to produce an updated stream.

It is a further object of the present invention to provide an apparatus and method for presenting a changing base file for an inverse write event journal as an unchanging base file.

Summary of the Invention

The present invention enables a broad collection of useful behaviors including operating with streaming media. Via the creation and use of an event map, the present invention is useful for more than backup and in particular includes a combination event journal and an ordered container. Through the use of an event map, the present invention enables, for example: an event log to be merged with a non-seekable stream; an event log to constitute a readable and writeable file; the use of only an event log to imitate a readable and writeable file; and an event log and a seekable readable file to imitate a readable and writeable base file. The present invention also supports use of streaming media in data protection applications previously restricted to random access media.

It is apparent that reliable and low-cost data protection is a formidable task. On one hand, conventional backup technology is cumbersome and the most recent data is always at risk. On the other hand, mirroring techniques instantly propagate errors to the backup storage, and large-scale deployment of mirroring is impractical because of hardware and costs. The present invention addresses these two major deficiencies of current data protection systems by providing up to the instant protection using low-cost media while causing minimum network transfer overhead.

Brief Description of the Drawings

Figure 1A illustrates a computer and computer storage devices according to an exemplary embodiment of the present invention.

Figure 1B illustrates exemplary storage units according to an embodiment of the present invention.

Figure 1C illustrates the effects of write events and addressing in a computer storage

device according to an exemplary embodiment of the present invention.

Figure 1D illustrates an exemplary event journal according to an embodiment of the present invention.

Figure 2A illustrates an exemplary event map according to an embodiment of the present invention.

Figure 2B illustrates an exemplary method for creating an event map according to an embodiment of the present invention.

Figure 2C illustrates an exemplary method for creating a current event marker according to an embodiment of the present invention.

Figure 2D illustrates an exemplary method for removing overlapped marker segments according to an embodiment of the present invention.

Figure 2E illustrates an exemplary method for revising an overlapped marker according to an embodiment of the present invention.

Figure 3 illustrates an original an updated storage according to an exemplary embodiment of the present invention.

Figure 4A illustrates exemplary components for fulfilling a read request according to an embodiment of the present invention.

Figure 4B illustrates an exemplary method for fulfilling a read request according to an embodiment of the present invention.

Figure 4C illustrates an exemplary method for building a stream according to an embodiment of the present invention.

Figure 5 illustrates an exemplary method for converting an event journal to a delta according to an embodiment of the present invention.

Figure 6A illustrates an exemplary flow chart showing how to use a read-only storage and an event journal as a seekable, readable and writeable storage according to an

embodiment of the present invention.

Figure 6B illustrates an exemplary flow chart showing how to write to a read-only storage and event journal combination according to an embodiment of the present invention.

Fig. 6C illustrates a read from a read-only storage and event journal combination according to an embodiment of the present invention.

Detailed Description of the Invention

Figure 1A shows a computer system 3 having, for example, two random access primary computer storages 1 attached via connection 2. The computer storages 1 are used, for example, to store computer data during normal operation of the computer system 3.

Figure 1B shows a more expanded view of a computer storage 1, showing elemental storage units 7, each having an address 5 and resident data, 10, e.g., "OLD_DATA_NOW". The storage addresses 5 for each elemental storage unit 7, for example shown having values 0-12, indicate the number of primary storage units the current storage unit is offset from the storage origin 4.

The elemental storage units 7 illustrated in Figure 1B are populated with, for example, ASCII byte codes. While the illustrated embodiment of the present invention recites fixed length storage units, a storage unit may contain a byte as illustrated, a disk block, a database record, or any other fixed length unit that satisfies the natural addressing convention of the particular computer. Storage units 7 may also be empty. During normal operation of the computer system 3, information is written to the elemental storage units 7. When new information is stored in a location, overwriting destroys the old information. In certain prior art systems, the old data is copied to a temporary area before overwriting occurs.

Figure 1C illustrates an original computer storage 1 containing original data 6. A collection of write events, for example indicated at 8, 9, 10, 11, 12 each result in a

change to computer storage 1 by overlaying data already there. For example, each write event including the data written and an address from the origin of the computer storage is indicated by 8-14, 9-17, 10-16, 11-18, 12-15 as shown in Figure 1C. The series of write events 13 resulting in a change in the data of the original storage 6 results in an updated storage 19.

Figure 1D illustrates an exemplary event journal. For the purposes of the present invention, the event journal may result from practice of any of the prior art methods recited above or an equivalent. Each event journal entry must include at least an event address 22 and the data recorded 24 during the write event. Note that number of elemental storage units written, or the event size 23, is a characteristic of the data written.

Each event entry in the event journal has an offset from the origin of the event journal. Each event entry also contains the data written to the original file. The data written to the original file also has an offset in the event journal. This offset is, for example, the event data address 25. This event data address is used to construct the marker data pointer later described. The event journal is organized as follows. Assuming, for example, that the event address 22 and the event size 23 are each 8 bit values, then the first event data 24, "abcdc", is stored at starting location 16, as shown in Figure 1D and would end at location 20. Also as shown in Figure 1D, the next event data, "fghi", is located at address 37 in the event journal which reflects the five address locations occupied by the first event data 24 plus the 16 bits occupied by the next event address 22 and event size 23. The remaining entries in the event journal are determined in the same manner.

For the purposes of simple illustration, all of the write events in this description apply to a single computer storage. The same methods also apply, however, to a computer or network of computers each having a plurality of storages. For systems having a plurality of storages, it is common to include checkpoint events. Checkpoint events contain markers that indicate stable or committed instants where data in a plurality of storages is synchronized or valid. Checkpoints often facilitate recovery to a particular

point in time for systems having a plurality of storages.

It is important to note that the events in the event journal are recorded in the same sequence that they occurred on the computer storage. Since this sequence is random with respect to the position in the original computer storage, there is no efficient way to determine the events which affected a particular storage offset without processing every event in the event journal.

The present invention uses an event map, or plurality of event maps, created from an event journal to ascertain the cumulative effects of a series of write events on original computer storage.

Figure 2A shows an event map according to an embodiment of the present invention. Figure 2B shows high level logic flow chart of an exemplary embodiment of the present invention for generating an event map. With reference now to Figures 2B-2E and source code pages 33-34 and 23-26 of the attached paper appendix and with particular reference to the source code routine entitled JournalServiceBase:RegisterEvent, the method according to the present invention proceeds as follows.

As indicated in Figure 2B, each event entry is loaded at step 32 and checked to see if it is a final event or a synchronization event indicating that the process should halt at step 33. If the event entry is not a final event, a current event marker is generated for the event at step 34. The event map is then searched for any marker segments that overlap the current marker and overlapping markers are removed at step 35. Finally, the current marker is inserted into the event map at step 36. The process continues until the last event is processed.

Figure 2C illustrates an exemplary method for constructing a current event marker from an event entry according to an embodiment of the present invention. As shown in Figure 2A, the current event marker comprises at least three components: a marker origin address 26 corresponding to the event data address of the loaded event entry; an event marker span 27 containing the number of primary storage units that were written

in the event entry; and a marker data pointer 28 comprising an address or offset in the event journal which enables the event data to be quickly located in the event journal.

For the purposes of illustration, exemplary generation of the event map is described from beginning to end. The method according to the present invention is useful if the event journal is stored on streaming media or if a backup computer is recording an event journal and simultaneously generating the event map.

Also according to the method of the present invention, an event map can be constructed by processing event journal entries in reverse order, from end to beginning. The mechanics are somewhat different because events encountered first, i.e., last in the journal, will take precedence over those earlier in the journal. The flow chart of Figure 2B would, instead of removing earlier events, first search for segments referenced in the event map. The overlapping segments found in the event map are omitted from the current marker. Note that the current marker can be fragmented into a plurality of markers each representing changes for the current event entry.

Figure 2D further describes an exemplary method for clearing overlapping markers from the event map according to an embodiment of the present invention. The event map is searched for the event closest to the current event marker in step 42. When no event is found, the event map is empty and there is no overlap with any other event and the method returns, at step 43, so that the current event marker can be inserted into the event map.

If an event is found, there are several possible conditions. The found event may start after the end of the current event in step 44. If this is the case, the previous event in the event map is loaded in step 45. If no such event entry exists in step 45, the method returns to step 43. Otherwise, the found event may end before the current event indicated in step 46. If this is the case, then no more events overlap the current event and then the method returns in step 50.

The final possible condition is overlap. Overlap occurs when part of the current marker sits on top of a found marker entry. When this occurs, the found marker entry

must be revised or removed to make way for the current marker entry in step 47. After the marker is revised in step 47, the previous event marker is loaded and the process continues until an exit condition is encountered in steps 43 or 46.

In a storage system, overlap occurs when a write occurs to the same location in a particular file. Many writes may occur to a file position resulting in many event entries referencing a particular storage location. Only the last write to the file position determines the data stored there. Equivalently, only the last event entry in the event journal defines the event marker for corresponding to that address in the event map.

The present invention uses a sorted container, such as a link list, array or btree, to contain the event markers. An example of a sorted container by Azarona Software employed in an exemplary embodiment of the present invention is included in the source code with particular reference to, for example, pages 105-117 of the attached paper appendix

Event markers are stored in order of event marker address. Sorting enables rapid location of markers relating to an event marker address. The process of inserting and deleting whole markers in the list can be time consuming, especially if the list or btree is large. The present invention practices two techniques for improved performance when lists become large. The first practice is known as marker editing. Editing modifies an existing entry in a list when it is known that the edits do not affect the sequence represented by the list. In most cases, editing an existing marker is many times faster than deleting and reinserting a tree entry or sorted list entry.

The practice of marker editing is particularly demonstrated, for example, in the Vtree::UpdateData routine particularly referenced on, for example, page 110 of the source code in the attached paper appendix. Marker editing techniques are further demonstrated in source code at, for example, page 26 line 368 in the attached paper appendix.

The second technique practiced according to the present invention is subdivision. It is well known in the art that the amount of effort to maintain a sorted container increases

disproportionally to the number of items in the list. The present invention can divide a large journal into a plurality of smaller segments and generate an event map for each journal segment.

There are several overlap conditions that can occur. A current event marker may overlap several markers, for example as illustrated in Figure 1C by events 8, 9, 10 overlapped by entry 12. A current marker may overlap a complete marker illustrated by event 10 being completely overlapped by event 12. A current marker may also overlap the trailing side of a current marker, illustrated by event 8 and event 12. A current marker may overlap the front side of a marker illustrated by event 9 and event 12.

Figure 2E illustrates an exemplary technique for revising an overlapped event marker according to the present invention. If the current event marker completely overlaps the found marker in step 52, the overlapped marker is deleted in step 53. If the current event marker overlaps the tail of the found event marker in step 54, the found event marker is rear trimmed by reducing the event data size represented in the found event marker in steps 55 and 56. If the front of the found event overlaps the found event in step 58, the found event is front trimmed by calculating the size of the overlap in step 59, increasing the event marker offset by the overlap in step 60, adjusting the marker data pointer to reflect the first data that was not overwritten in the event journal by adding the overlap to the marker data pointer in step 61, and finally reducing the data size of the found marker in step 62.

It is considered within the scope of the present invention to use marker editing techniques to replace a marker entry deletion and insertion when the deleted and inserted markers go into the same place in the sorted container.

The event map according to the present invention is useful for a variety of purposes. It is well known to create a backup of a computer by copying an original storage to a streaming media. Prior art systems describe methods for recovering from an event journal by copying a backup onto a hard disk and "replaying" the events in an event

journal. This technique only works, however, if the number of events in the event journal is small enough to replay in a reasonable amount of time. For example, if a large volume of changes are stored in an event journal, replaying the entire event journal to recreate a file could take a prohibitively long time. Thus, such a technique is impractical for sustained off-site backup maintenance. The requirement to periodically refresh the entire backup creates a huge amount of network traffic and disqualifies this method from use for large systems. In addition, the prior art systems require the intermediate step of placing the original data file on a seekable medium prior to replaying the event journal to recreate a file. On conventional tape back-up systems, however, only a small amount of disk space is available, if at all, and thus the original file cannot be placed on disk for merging with an event journal as is done via an event map according to the present invention.

In contrast, the event map of the present invention enables efficient updating of a backup stored on streaming media. For example, by creating the event map according to the present invention, the net result of the changes in the event journal are combined with the original file, thus reducing the amount of network traffic associated with the back-up or recreation process and there is no requirement for an intermediate step of placing the original file on a seekable medium as the event map can be combined sequentially with the original file.

As indicated earlier, it is well known in the art that a backup comprises a copy of an original storage and that backup copies are often stored on streaming media because streaming media is cheaper than random access media. It is also well known to store an event journal. For example, in a conventional computer system with a primary computer and a backup computer connected to a network, a copy of a base (e.g., original) file is copied from the primary computer to the backup computer. To generate a backup copy of the current state of the file, the base file would be written to a disk from the backup computer for combination with the changes to the base file, stored as an event journal on the backup computer, thus necessitating many I/O operations as described earlier. The updated file would then be stored in the backup

system. In addition to requiring transfer from a backup streaming media to a disk to generate an updated backup copy of a file, such a backup system also generally does not provide the capability to incorporate only recent changes to the base file, which may no longer exist on the backup system if replaced following a backup operation. Thus, it is not known to merge data in an event journal with an original storage stored on streaming media for recovery. It is further not known to merge data from an event journal with data in a streaming media for maintenance of a backup copy to keep the backup copy up to date in accordance with the present invention.

It is known in the art to replay recorded storage events. Replay techniques repeat the sequence of writes recorded in an event journal to a copy of an original storage stored on random-access random-writeable media. A problem with this approach is that the vast majority of the copies of original-storage are stored on streaming media, e.g., tape, which is neither efficiently seekable nor randomly writeable. As a result, event journal techniques are not used for backup.

The present invention enables efficient merging of information in an event journal with a copy of an original storage on streaming media. The methods described above relating to creation of an event map, when practiced with the following techniques to fulfill a data request from an event journal and an original storage, enable an array of new capabilities. Further disclosure of this technique is provided with reference to, for example, pages 23-26 and 33-34 of the source code in the attached paper appendix and with particular reference to the JournalServiceBase::QueryLocation subroutine.

Figure 4A illustrates exemplary components of the present invention that participate in fulfilling a read request 30 for an updated storage, an original storage 6, an event journal 21 and an event map 29. The flowchart of Figure 4B describes an exemplary method for fulfilling a read request from the combination of Figure 4A comprising an original storage 6, an event journal 21 and an event map 29. A read request is composed, for example, of two elements: a data position; and a read size. The data position gives, for example, the starting address relative to an origin of the data to be

read. The read size gives, for example, the count of elemental units to be obtained from the storage. The sum of the data position and the read size gives the address of the ending read address.

With reference to Figure 4B, the first step 66 in processing a read request is to determine the data position, read size, and ending read address 32. The event map is queried for a marker that contains the current read position in step 67. If no marker references the current read position in step 68, the number of storage units until the next read marker is retrieved in step 72, the next marker count. The unit read size is calculated to be the minimum of the next marker count and the read size in step 73. Data from the original storage is copied into the read buffer to fulfill the unit read count of primary storage elements in step 74.

If an event marker is found which corresponds to the read position in step 68, the unit read size is calculated to be the minimum of the overlapping marker segment size and the read size in step 69. The marker data pointer is used to locate the corresponding event data in the event journal and fulfill the request for unit read size from the event journal in step 70. Next, the read size is decremented by the number of elemental storage units fulfilled, unit read size, in the last iteration, and the read position is advanced by the unit read size to indicate partial fulfillment of the read request in step 71. When the read size reaches zero, the read is fulfilled in step 75 and the process terminates in step 76. If the read size is not zero, the process resumes by querying the event map in step 67.

Application of the read method according to the present invention to cause sequential reading of an updated stream from beginning to end is an efficient way to merge an original stream and an event journal. Figure 4C shows an exemplary flow chart that generally describes the method according to the present invention. This method is further disclosed in source code form at, for example, pages 9-10, 15-18, 19-20, 23-26, 30-31 and 33-34 in the attached paper appendix. The source code references four similar but distinct uses of the present invention. Each of these several behaviors can

be, for example, invoked by program options.

As illustrated in Figure 4C, in a method for merging a non-seekable base stream with an event log, an event map is constructed from the event journal as described above in step 78. A copy of an original storage on a streaming media is loaded into a tape drive in step 79. A series of read requests requesting consecutive segments of data from the updated storage represented by the combination of the original storage and the event journal are issued and fulfilled by, for example, the method of Figure 4B in step 80. The results of the read requests are subsequently recorded to a target storage which may be another streaming media, disk or other storage in step 81. The process continues until complete in steps 82 and 83.

The sequential read process, the subject of Figure 4B and source code disclosure at, for example, page 20 of the attached paper appendix, causes the copy of the original storage to be consumed from beginning to end. The seeks which occur to the original storage advance the original storage beyond the same number of primary storage units supplied by the event journal. As a result, the original storage is consumed from beginning to end without seeking notwithstanding skipping of data units provided by the event journal. This characteristic enables efficient combination of an original computer storage and an event journal.

Seeks on the original storage serve to skip data segments provided by the event journal. Co-pending application by applicant, entitled COMPUTER APPARATUS AND METHOD FOR MERGING A SEQUENTIAL PLURALITY OF DELTA STREAMS recites a method and apparatus to capture skipped segments into an inverse delta. When processing a stream from beginning to end, the act of discarding characters is awkward. In practice, seeks in the original media only occur when a segment from the original storage has been overwritten. The normal effect of this on an original stream is to skip the overtyped characters. As recited above, the means of skipping overtyped characters is to discard them. The present invention includes, for example, methods compatible with co-pending application entitled COMPUTER APPARATUS AND METHOD FOR MERGING A SEQUENTIAL PLURALITY OF

DELTA STREAMS which, for example, captures an "inverse delta" which is a list of changes that if made to the updated storage convert it back into an original storage. The present invention also produces inverse deltas. The method simply requires capturing elemental storage units skipped in the original stream as mismatch segments and recording data segments used from the original stream as matching segments.

It is similarly an object of the present invention to translate an event journal into a delta. A delta contains, for example, alternating frames describing matching and mismatching sections of an original and updated storage. The method is disclosed in source code with reference to, for example, pages 12, 21-22, 32, and 33-34 of the attached paper appendix with particular reference to the class named JournalDelta.

The flow chart of Figure 5 illustrates an exemplary embodiment of a method for converting an event journal to a delta according to the present invention. An event map is constructed for the event journal in step 85. A variable tracking the logical progress through the updated stream is initialized in step 87. This variable tracks the position accounting of the updated file. This logical position advances resulting from an accounting for storage units in the updated stream. Each time this position advances, the curposition variable is advanced in step 94.

When the curposition variable reaches a known EOF condition, the method terminates in steps 88 and 95. For all other times, the event map is queried for the curposition in step 89. When the query returns with a match marker notification, the match marker is used to construct a data frame. The data frame specifies a mismatch in the original and updated storages. The data frame comprises effectively the data that was not matched. The mismatching data is extracted from the event journal using the marker data pointer and the marker data size and incorporated into the data frame.

When the query returns no match marker notification in step 90, the data from the original and updated streams are identical until the next match marker. In an embodiment of the present invention, a count of primary storage units is returned until the event address of the next event marker. The curposition variable and this count

are used to construct the match frame in step 93. The match frame tells the position and count of characters that match in the original and updated storage. The match frame notification comprises a position element and a size element indicating the position and number of characters that match in the original and update streams.

Finally, the generated frame is recorded in step 94 and the process resumes. The curposition is incremented to account for data represented by the current frame in step 94 and the process resumes by checking if the storage is complete in step 88. If not, the process above repeats until all elemental storage units of the updated storage are accounted for. The delta of the present method is particularly useful when used in conjunction with co-pending applications entitled COMPUTER APPARATUS AND METHOD FOR MERGING A SEQUENTIAL PLURALITY OF DELTA STREAMS and COMPUTER APPARATUS AND METHOD FOR MERGING SYSTEM DELTAS.

Read-only files are well known in the art. They are common to write-once media such as CD-ROMs and the like as well as network file systems where a user may lack permission to or the ability to modify a particular storage. The present invention further provides a means of using a combination of a read-only storage, an active event journal and an event map as a seekable-readable-writeable storage. The method is generally disclosed in source code at, for example, pages 9-10, 15-18, 19-20, 23-26, 30-31 and 33-34 of the attached paper appendix. The flowcharts of Figure 6A-6C generally describe an exemplary method according to the present invention.

The method of Figure 6A includes, for example, the step of initializing an event journal in step 97. Initialization may be, for example, creation of a new event journal or activation of an existing journal. If the session refers to a continuation of an earlier session, the storage associated with the event journal is opened for reading and writing. If this is a new session, an event map is created, otherwise an earlier event map is activated in step 98. The event map and the event journal should be consistent. Note that if an event journal exists but no event map exists, the above method for generating an event map from an event journal is used. The final step is to open the

read only storage in step 99. Note that by definition the read-only cannot be modified.

After initialization of the event journal, event map and read only storage, read and write accesses to the storage are performed as generally described in step 100, and specifically performed as described in Figures 6B and 6C. With further reference to Figure 6B, the present invention diverts writes that would normally apply to the read-only storage to the event journal. This diversion is performed by first constructing a write event entry from the write request by determining the data and position represented by the write request. The data position is used as the event address. The data included in the write request is used as the event data.

The write event entry is recorded into the event journal in step 105. Subsequently, the event entry is used to construct a current event marker in step 106 using, for example, the method generally represented in Figure 2C. The event map is searched and all overlapping segments that overlap the current event marker are removed in step 107 using, for example, the method generally represented in Figure 2D. Finally, the current event marker is added to the event map in step 108.

Read requests to the combined read-only storage, event journal combination are generally processed using the method illustrated, for example, in the flowchart of Figure 6C. Instead of reading the source file, the read request is diverted and fulfilled by the method generally represented by Figure 4B. The combination of using this read and write method provides a readable and writeable interface to a read-only storage.

The technique according to the present invention can be used, for example, to provide a plurality of interfaces to a read-only file. Consider, for example, a group of users all having access to a read-only storage but desiring to make changes to this storage. The method according to the present invention can be applied for each user who generates an independent event log that contains only the changes made by the user. These changes are invisible to the other users permitting each user to change his data view as necessary.

A similar application of the present invention uses the above method for simulation of a standard file interface using only a read only original storage and an event log in the absence of the read only file. If the read-only file above contains no data then the event journal contains all of the subject data. This capability permits a readable writeable and seekable file system to be created on a seekable write-once media like a CD-ROM. The method involves creating the event journal on CD-ROM and using the read and write simulation methods disclosed in the previous section to fulfill all read and write requests.

JOURNAL INDEXING SERVICES

SOURCE CODE SECTION

```
 1 /*
 2   This file contains the base class interface to the
 3   base stream.  These the methods which will be overridden
 4   to adapt this object to a new base stream type.
 5
 6   The NOWRITE mode causes blocks changes from the base file
 7   from being visible by recording inverse events into the
 8   event journal.  Therefore open the base file with write
 9   permission when NOWRITE mode is set.
10 */
11 #include "jfile.h"
12
13 void Jfile::OpenBaseStream( char* BaseFile ) {
14   // The basefile is optional because
15   // the journal file may contain everything
16   if ( BaseFile && strlen( BaseFile ) ) {
17     if ( JfileMode & BASE_DYNAMIC ) {
18       // We'll be writing to the base in dynamic mode
19       BaseFD     = open( BaseFile, O_RDWR | O_BINARY );
20     } else {
21       BaseFD     = open( BaseFile, O_RDONLY | O_BINARY );
22     }
23     if ( BaseFD == -1 )
24       throw FileError( "Jfile::Open", BaseFile );
25     else {
26       // !BASE_SEEKABLE is the default
27       BaseStreamPosition = 0;
28     }
29     // We need to know how large the base file is
30     if ( JfileMode & BASE_SEEKABLE ) {
31       BaseFileSize = lseek( BaseFD, 0, SEEK_END );
32       // Rewind to the beginning
33       lseek( BaseFD, 0, SEEK_SET );
34     }
35   }
36   if ( BaseFile )  BaseFileName = BaseFile;
37
38 }
39
40 void Jfile::CloseBaseStream() {
41   if ( BaseFD > 1 ) {
42     close( BaseFD );
43     BaseFD = -1;
44   }
45 }
46
47 int Jfile::ReadBaseStream( void* Buf, int Cnt ) {
48   if ( BaseFD == -1 && Cnt > 0 ) {
49     throw ProgramError( "Jfile::ReadBaseStream"
50                         "Attempted to read with no basestream open" );
51   }
52   int BaseGotCnt = Cnt;
53   // Constrain the read to the logical end of the base stream
54   // this is necessary if we are in base_dynamic mode because
55   // NotWrite expects the read to stop at the old end of file,
56   //   multiple calls may extend EOF beyond the old one, so
57   //   you have to enforce EOF on reads.
58   if ( BaseFileSize > -1 ) {
59     if ( BaseGetCnt > BaseFileSize - BaseStreamPosition )
60       BaseGetCnt = BaseFileSize - BaseStreamPosition;
```



```
61   }
62   int BaseGotCnt = read( BaseFD, (char*)Buf, BaseGetCnt );
63   BaseStreamPosition += BaseGotCnt;
64   return BaseGotCnt;
65 }
66 /*
67    Note that this function is only used to demonstrate
68      the naturite method.
69
70
71 */
72 int Jfile::xWriteBaseStream( void* buf, int Cnt ) {
73 //  SeekBaseStream( CurPosition );
74   int Stat = write( BaseFD, buf, Cnt );
75   return Stat;
76 }
77 /*
78   This method positions the base file at a position.
79
80   The basefile may be set to non-seekable mode, ie.
81   a stream.  If this is the case, then this function
82   should burn characters between the BaseStreamPosition
83   and the target location.
84
85   If the basefile is seekable, then a standard seek is okay.
86 */
87
88 void Jfile::SeekBaseStream( long Position ) {
89   // No base file is open, then return eof
90   if ( BaseFD == -1 ) {
91     BaseStreamPosition = EOF;
92     return;
93   }
94
95   if ( JfileMode & BASE_SEEKABLE ) {
96
97     if ( Position < BaseFileSize ) {
98       BaseStreamPosition = lseek( BaseFD, Position, SEEK_SET );
99     } else {
100      // Beyond EOF
101      BaseStreamPosition = lseek( BaseFD, BaseFileSize, SEEK_SET );
102      BaseStreamPosition = EOF;
103    }
104  } else {  // Base Not Seekable
105
106    // Don't use BaseFileSize here because we probably don't know it
107    if ( Position < BaseStreamPosition ) {
108      throw ProgramError( "Jfile::SeekBaseFile",
109                          "Cannot seek backwards in non-seekable file" );
110    }
111
112    // Burn enough characters to service the
113    // seek.  Discard them using the discard hook
114    long AdvanceCnt = Position - BaseStreamPosition;
115    Bufr B( MAXBUF );
116
117    // Discard characters
118    while ( AdvanceCnt ) {
119      int GetCnt = min( MAXBUF, AdvanceCnt );
120      int GotCnt = read( BaseFD, B, GetCnt );
```

```
121     DiscardBaseStreamSegment( B.Ptr(), GotCnt );
122     BaseStreamPosition += GotCnt;
123     AdvanceCnt -= GotCnt;
124
125     // We hit eof
126     if ( GotCnt != GotCnt ) {
127       BaseStreamPosition = EOF;
128     }
129   }
130 }
131 }
132
133 void JFile::ForceSeekBaseStream( long Position ) {
134   BaseStreamPosition = lseek( BaseFD, Position, SEEK_SET );
135 }
```

C:\DELTAVPATENT\LOGFILE\ERICQBASESTRM.CPP

Mon Sep 30 13:40:46 1996

```
1 /*
2   delta.cpp
3
4   This file implements the methods which package
5 an event journal as a delta file.
6 */
7
8 /*
9   This method fills out the frame header for the
10  next frame and inserts the appropriate data into
11  the data buffer
12 */
13
14 JournalDelta::JournalDelta() : DataBuffer( MAXDATAFRAME )
15 {
16
17 }
18 JournalDelta::~JournalDelta() {
19 }
20 int JournalDelta::GetFrame( FrameHdr& FH, void* Data )(
21
22 }
```

```
 1 /*
 2   deltmain.cpp
 3
 4   This main file uses the DeltaJournal object to produce
 5   a series of frames for a delta file.
 6
 7 */
 8
 9 #include <jrndelta.h>
10 #include <cmdline2.h>
11 #include <iostream.h>
12 #include <fstream.h>
13 #include <sys/stat.h>
14
15 int main(int argc, char *argv[])
16 {
17   CmdLine2 CL;
18   CL.Init( argv );
19
20   if ( argc < 2 ) {
21     cout << "deltajrn <options>" << endl;
22     cout << "    jf=<jrn>          Journal File Name" << endl;
23     cout << "    startcmd=string   Run this file before starting" << endl;
24     cout << "    yack              Diagnostic output" << endl;
25     cout << "    home=<dir>        Use working directory" << endl;
26     cout << "    basesize=<-1>     Provide a size for the base stream" << endl
   ;
27     cout << "                        (required from eventjrn or manual)" << en
   dl;
28     return 0;
29   }
30
31   if ( CL.FindHome() ) {
32     CL.GoHome();
33     cout << "Working Directory:  " << CL.GetHome() << endl;
34   } else
35     cout << "Working Directory:  " << CL.GetHere() << endl;
36
37
38   if ( CL.Find( "startcmd" ) ) {
39     Sbuf StartCmd;
40     CL.getline( StartCmd.Ptr(), StartCmd.size(), 0 );
41     system( StartCmd.Ptr() );
42   }
43
44   Sbuf JournalFileName = "jrn";
45   if ( CL.Find( "jf" ) ) {
46     CL.getline( JournalFileName.Ptr(), JournalFileName.size(), 0 );
47   }
48
49   long BaseStreamSize = -1;
50   if ( CL.Find( "basesize" ) ) {
51     CL >> BaseStreamSize;
52   }
53
54   try {
55     // Share a Jfile instance
56     JournalDelta JD;
57
58     JD.Open( JournalFileName );
```



```
59     FrameHdr FH;
60
61     // If the event file did not provide a base stream
62     // size use the one from the command
63     if ( JD.GetBaseStreamSize() == -1 ) {
64       JD.SetBaseStreamSize( BaseStreamSize );
65     }
66     void *DataBuf = NULL;
67     do {
68       int GotCnt = JD.ReadFrame( FH, DataBuf );
69       if ( GotCnt <= 0 ) break;
70
71       FH.printOn( cout );
72       switch ( FH.Key ) {
73         case T_MATCH:
74           ((MatchFrame*) DataBuf)->printOn( cout );
75           break;
76         case FIRST:
77           ((FirstFrame*) DataBuf)->printOn( cout );
78           break;
79         case LAST:
80           ((LastFrame*) DataBuf)->printOn( cout );
81           break;
82         case DATA:
83           cout.write( (char*) DataBuf, FH.Len );
84           cout.flush();
85           break;
86         default:
87           cout << "Error Unrecognized frame" << endl;
88           return -1;
89       }
90     } while ( FH.Key != LAST );
91
92
93   } catch ( Exception& EX ) {
94     char* Report = EX.GetReportString();
95     return -1;
96   }
97   return 0;
98 }
```

```
 1 /*
 2   Test program to for the composite key comparison routines
 3 */
 4
 5 #include <iostream.h>
 6 #include <string.h>
 7 #include <stdlib.h>
 8 #include "rela2.h"
 9 #include <cmdline2.h>
10 #include <time.h>
11
12
13 extern _fmode = O_BINARY;
14
15 /*
16    I increased the key size to increase the probability
17    if internal boundary conditions
18
19    At worst case 2048 / 128 = 16 keys should fit into the node
20 */
21
22 main(int argc, char *argv[])
23 {
24   CmdLine2 CL;
25   CL.Init( argv );
26   cout << "Working Directory:  " << CL.GetWere() << endl;
27
28   if ( argc < 2 ) {
29     cout << "ix <options>" << endl;
30     cout << "    fname=<test.idx>" << endl;
31     cout << "    create       Create a new index" << endl;
32     cout << "    delrandom    Delete a series of random keys" << endl;
33     cout << "    open         Open an existing index" << endl;
34     cout << "    list         List the contents of an index" << endl;
35     cout << "    stat         List statistics for index" << endl;
36     cout << "    addrandom    Add a set of random keys" << endl;
37     cout << "    seed=<20>    Use seed to generate random keys" << endl;
38     cout << "    keycnt=<3>   Number of keys to generate" << endl;
39 //    cout << "    ixcnt=<1>    Number of indexes in the file" << endl;
40     cout << "    keysize=<20> Maxsize of keys to generate" << endl;
41     cout << "    block=<512>  Size of index block" << endl;
42     cout << "    yack         Echo key content before action" << endl;
43     cout << "    timer        Report time of process" << endl;
44 //    cout << "    revert       Revert index to original state" << endl;
45     return 0;
46   }
47
48   Sbuf Home;
49   if ( CL.Find( "home" ) ) {
50     CL.getline( Home.Ptr(), Home.size(), 0 );
51     CL.GoHome();
52   }
53
54   long StartTime = time( NULL );
55   unsigned seed = 20;
56   if ( CL.Find ( "seed" ) ) CL >> seed;
57
58   long Ntry = 3;
59   if ( CL.Find ( "keycnt" ) ) CL >> Ntry;
60
```



```
61   int UserKeySize = 20;
62   if ( CL.Find ( "keysize" ) ) CL >> UserKeySize;
63   int RealKeySize = UserKeySize + Entryb::BaseEntrySize();
64
65   int CacheSize = VTREE_DEFAULT_CACHESIZE;
66   if ( CL.Find ( "cache" ) ) CL >> CacheSize;
67
68   int MaxHeight = VTREE_DEFAULT_MAXHEIGHT;
69   if ( CL.Find ( "maxheight" ) ) CL >> MaxHeight;
70
71   int BlockSize = 512;
72   if ( CL.Find ( "block" ) )
73     CL >> BlockSize;
74
75   // Check the block sizing relative to the key size
76   if ( BlockSize <= 3 * RealKeySize ) {
77     BlockSize = 6 * RealKeySize;
78     // Round to a 512 byte size
79     BlockSize += 512 - ( BlockSize % 512 );
80     cout << "Increasing Tree Block Size to:  " << BlockSize << endl;
81   }
82
83   int Yack = 0;
84   if ( CL.Find( "yack" ) ) Yack = 1;
85
86   Vtree VT;
87
88   Sbuf Fname;
89   Fname = "test.idx";
90   if ( CL.optA( "fname" ) )
91     CL.getline( Fname.Ptr(), Fname.size(), 0 );
92
93 try {
94
95   if ( CL.optA( "create" ) ) {
96
97     VT.Create( Fname );
98     // Root.Create( );
99     // Set up the default relations
100    char kv[4]; memset( kv, 0, 4 );
101    kv[0] = Entryb::CHAR;
102    VT.SetKeyVector( kv );
103
104  } else {  // Load an index
105    int Stat = VT.Open( Fname, Vtfile::READ_WRITE );
106    if ( Stat == VT_FAIL ) {
107      cerr << "Index Open Failed:  " << Fname << endl;
108      return -1;
109    }
110  } // End of create load pair
111
112  // Entryb E( VT );
113
114  if ( CL.optA( "addrandom" ) ) {
115    // Number of entries to generate
116    srand(seed);
117    cout << "Seed:              " << seed << endl;
118    cout << "Indexing items:  " << Ntry << endl;
119
120    for ( long i = 0; i < Ntry; i++) {
```

```
121
122     // Allocate and clear key buffer
123     char* KeyBuf( UserKeySize + 1 );
124
125     // Generate a random length
126     // Leave space for the long key
127     int rlen = rand( ) % ( UserKeySize + 1 ) - 2;
128
129     // fill the KeyBuf with random data
130     char* WrBuf = KeyBuf;
131     for ( ; rlen > 0 ; rlen--) {
132       int RandomLetterIx = rand( ) % 26;
133
134       *WrBuf = RandomLetterIx + 'A';
135       WrBuf ++;
136     }
137
138     // Add the first key segment
139     EntryD EE( VI );
140     EE.AddKeyElement( KeyBuf.Ptr() );
141     long NowTime = time( NULL );
142     EE.data_field = NowTime;
143
144     if ( Vack ) { cout << EE.Size() << endl;
145                   EE.printOn( cout ); }
146     // Add the key to the index
147     VI.Add( EE, 1);
148   } // End of adding keys loop
149 } // Addrandom
150 if ( CL.optAt( "delrandom" ) ) {
151   // Number of entries to generate
152   srand(1966);
153   cout << "Seed:          " << seed << endl;
154   cout << "Deleting Items: " << Entry << endl;
155
156   for ( long i = 0; i < Entry; i++) {
157
158     // Allocate and clear key buffer
159     Bufr KeyBuf( UserKeySize + 1 );
160
161     // Generate a random length
162     // Leave space for the long key
163     int rlen = rand( ) % ( UserKeySize + 1 );
164
165     // fill the KeyBuf with random data
166     char* WrBuf = KeyBuf;
167     for ( ; rlen > 0 ; rlen--) {
168       int RandomLetterIx = rand( ) % 26;
169
170       *WrBuf = RandomLetterIx + 'A';
171       WrBuf ++;
172     }
173     // Add the first key segment
174     EntryD EE( VI );
175   *EE.AddKeyElement( KeyBuf.Ptr() );
176
177     EE.data_field = i;
178
179     if ( Vack ) EE.printOn( cout );
180     // Add the key to the index
181   if ( VI.Delete( EE, 1) != VI_SUCCESS ) {
182     cout << "Key not found" << endl;
183   }
184 } // End Of key loop
185 } // Delrandom
186 if ( CL.optAt( "list" ) ) {
187   cout << "Listing Index: " << endl;
188   VI.BeforeAllEntry();
189   EntryD EE( VI );
190   while ( VI.Forward() == VI_SUCCESS ) {
191     VI.GetCurr( EE );
192     EE.printOn( cout );
193   }
194 }
195
196 if ( CL.optAt( "stat" ) ) {
197   VI.Statistics( 1 );
198 }
199
200 VI.Close();
201 if ( CL.FindX "time" ) ) {
202   cout << "Run Time:  " << time( NULL ) - StartTime << endl;
203 }
204
205 } catch ( ExceptionX EX ) {
206   char* ReportString = EX.GetReportString();
207   return -1;
208 }
209 return 0;
210 }
211
```

```
 1 /*
 2   Jfile.cpp implements a file-like object with standard
 3   read, write, seek interfaces.  Except jfile does not
 4   use a standard file.
 5
 6   It uses an index and a base file to emulate a standard
 7   file interface.
 8 */
 9 #include "jfile.h"
10 #include <sys/stat.h>
11
12 Jfile::Jfile() {
13   BaseFD      = -1;
14   JournalFD   = -1;
15   CurPosition = -1;
16   CurLength   = -1;
17   BaseStreamPosition = -1;
18   JfileMode = 0;  // Seekable for now
19   BaseFileSize = -1;
20 }
21 /*
22   This method prepares a basefile and a journalfile
23   pair.  If no index exists, one is created.  For now the
24   index is JournalFile.I
25 */
26 void Jfile::Open( char* JournalFile, char* BaseFile ){
27
28   OpenBaseStream( BaseFile );
29   JournalServiceBase::Open( JournalFile );
30   /*
31   JournalFD = open( JournalFile, O_RDWR | O_BINARY );
32
33   if ( JournalFD == -1 ) {
34     JournalFD = open((const char *)JournalFile,
35           O_CREAT | O_TRUNC | O_RDWR | O_BINARY,
36           S_IREAD | S_IWRITE );
37   }
38   if ( JournalFD == -1 )
39     throw FileError( "Jfile::Open", JournalFile );
40
41   JournalFileName = JournalFile;
42   JournalIxName   = JournalFile;
43
44   // Add a .I to the end of the journal name
45   char *Ext = strchr( JournalIxName, '.' );
46   if ( Ext ) strcpy( Ext, ".I" );
47   else JournalIxName += ".I";
48   int Stat;
49   try {
50     Stat = VT.Open( JournalIxName, VTFile::READ_WRITE );
51   } catch ( FileError& FE ) {
52     FE = FE;
53     Stat = VT_FAIL;
54   }
55   if ( Stat == VT_FAIL ) {
56     Stat = VT.Create( JournalIxName );
57     // Set up the key format for the tree
58     char kv[6];
59     memset( kv, 0, 4 );
60
```



```
61     kv[0] = EntryB::INT32;  // Offset in base file
62     // Install the key template into the tree
63     VT.SetKeyVector( kv );
64
65     // Build an index for the journal
66     IndexJournalFile();
67   }
68   if ( Stat == VT_FAIL ) {
69     throw FileError( "Jfile::Create", JournalIxName );
70   }
71   */
72   CurPosition = 0;
73 }
74
75 void Jfile::Close() {
76   // Shut down the tree
77   //VT.Close();
78   CurPosition = -1;
79   CurLength   = -1;
80   CloseBaseStream();
81   //close( JournalFD );
82   //JournalFD = -1;
83   JournalServiceBase::Close();
84 }
85 //
86 // This method retrieves a buffer of data from
87 // the jfile.
88 //
89 int Jfile::Read( void* Buf, int Cnt ) {
90   EventEntry EE;
91   int GotCnt = 0;
92   //int BufPosition = 0;
93   char* B = (char*)Buf;
94   while ( Cnt != GotCnt ) {
95     __int64 Stat = QueryLocation( CurPosition, EE );
96
97     // Retrieve from event file
98     if ( Stat == 0 ) {
99       // Note that if there was no base file, QueryLocation
100      // will always return an event, except when we have
101      // asked for something beyond EOF.
102
103      // Go to the data position in the journal file
104      lseek( JournalFD, EE.Position, SEEK_SET );
105      // How many characters should be retrieved from the event
106      long EvtGetCnt = min( Cnt - GotCnt, EE.Span );
107      int EvtGotCnt = read( JournalFD, B + GotCnt, EvtGetCnt );
108      if ( EvtGotCnt != EvtGetCnt ) {
109        throw FileError( "Jfile::Read", JournalFileName );
110      }
111      // Update the read status variables
112      GotCnt += EvtGotCnt;
113      CurPosition += EvtGotCnt;
114    } else { // Stat > 0 or Stat == -1
115      // Retrieve from base file
116      // Seek the base file to the proper location
117      SeekBaseStream( CurPosition );
118
119      if ( BaseStreamPosition == EOF ) {
120        // An event might have advanced CurPos beyond
```

```
121     // the end of the base file.  If so this is it.
122     return GotCnt;
123   }
124
125   int64 BaseGotCnt;
126   if ( Stat > 0 ) {
127     // Not EOF, read from the basefile until the next event
128     BaseGotCnt = min( Stat, BaseGetCnt );
129   } else { // Stat == -1
130     // EOF then ask for the unfulfilled amount
131     BaseGotCnt = Cnt - GotCnt;
132   }
133
134   int BaseGotCnt = ReadBaseStream( B + GotCnt, (int) BaseGetCnt );
135
136   // Update the read status variables
137   GotCnt += BaseGotCnt;
138   CurPosition += BaseGotCnt;
139   // BaseStreamPosition += BaseGotCnt;
140   // If this was a short read, we hit EOF on
141   // the Basefile.  Don't try to read anymore
142   if ( BaseGotCnt != BaseGetCnt ) {
143     break;
144   }
145 } // If Stat != 0
146 } // End while
147 return GotCnt;
148 }
149
150 /*
151 This method queries a location and returns relevant
152 information.
153
154   1) If the location is within an event, the EE
155      structure is filled and zero is returned.
156
157   2) If the location is not in an event, the
158      number of characters until the next event
159      are returned.
160
161   3) If there are no subsequent events, then -1 (EOF)
162      is returned.
163
164   //   Example 2:
165   //
166   //   012345   012345
167   //   1++++_ => 1+
168   //            2+
169   //                    1  // 3 events
170   //
171   //
172 */
173
174 int64 JFileIsBusyLocation( __int64 Location, EventEntry& EE )
175 {
176   // Construct a search entry
177   EntryB SearchKey( VT );
178   SearchKey.AddKeyElement( (long) Location );
179
180   // Go to the exact key or next larger or end of tree
_equiBar_____C:\DELTAVPATIENT\LOGFILES\SRC\JFILE.CPP_____

181   int Stat = VT.FindLeast( SearchKey );
182   if ( Stat != VT_FAIL )
183     return -1;  // No Events in Journal
184
185   SearchKey.printOn ( cout );
186
187   // Load the starting position of the found key
188   long FoundKeyBegin; short Val;
189   SearchKey.GetKeyValue( 0, &FoundKeyBegin, Val );
190
191   // If we found an exact match then return
192   // the entry information
193   if ( FoundKeyBegin == Location ) {
194     SearchKey.GetData( &EE, sizeof( EventEntry ) );
195     return 0;  // Event Found
196   }
197
198   // Otherwise the found event is beyond the location
199   // Remember the space between the location and
200   // the found event
201   __int64 DistanceToNextEvent = FoundKeyBegin - Location;
202
203   // Now Back up by an event
204   // If this is the last entry in the tree
205   // We might be within its span
206   if ( FoundKeyBegin > Location ) {
207     Stat = VT.BackWard( SearchKey );
208     if ( Stat == VT_FAIL ) {
209       // No earlier event tell caller how far to next one
210       return DistanceToNextEvent;
211     }
212   }
213
214   // So we should be at the event just before the location
215   // Check to see if location is within the event
216   SearchKey.printOn ( cout );
217
218   // Load the starting position of the found key
219   // long FoundKeyBegin; short Val;
220   SearchKey.GetKeyValue( 0, &FoundKeyBegin, Val );
221
222   EventEntry EE;
223   SearchKey.GetData( &EE, sizeof( EventEntry ) );
224   long FoundKeyEnd = FoundKeyBegin + EE.Span;
225
226   // If Location is beyond the earlier event,
227   // then location is not within an event
228   // Report distance to next event
229   if ( Location >= FoundKeyEnd ) {
230     if ( DistanceToNextEvent < 0 ) return -1;
231     else return DistanceToNextEvent;
232   }
233
234
235   // Otherwise Location resides within an event
236   // Build a EventEntry to notify the caller
237   // of both the offset and span and offset
238   // of the info available in the journal file
239
240   //  0123456789012345
```

```
241   //  1++++ 2++ 3++
242   //
243   //  Location = 3
244   //  SearchKey = 0_pos,4_span
245   //  JournalFileSpan   = How much span after location
246   //  JournalFileOffset = OldJrnPosition + (OldSpan - JournalFileSpan)
247   EE.Span       = FEE.Span - (Location - FoundKeyBegin);
248   EE.Position   = FEE.Position + ( FEE.Span - EE.Span );
249   return 0;
250 }
251 */
252
253
254 /*
255   The seek method places the Jfile at a particular location
256   in preparation for reading or writing.  You cannot seek
257   beyond the end of the Jfile.
258
259 */
260
261 __int64 Jfile::Seek( __int64 NewPosition ) {
262   //CurPosition = NewPosition;
263
264   //  Note that with non-seekable basefile there is no way to
265   //  query how large the actual file is.
266   if ( BaseFileSize == -1 ) {
267     CurPosition = NewPosition;
268     return CurPosition;
269   }
270
271   // If we know that the target is less than basefilesize
272   if ( NewPosition <= BaseFileSize )  {
273     CurPosition = NewPosition;
274     return CurPosition;
275   }
276
277   // Otherwise we need to check the event file
278   EntryD SearchKey( VT );
279   VT.AfterMaxEntry( );
280   int Stat = VT.Backward( SearchKey );
281   if ( Stat == VT_FAIL )
282     throw ProgramError( "Jfile::Seek",
283                         "Cannot Position after all data" );
284
285   long FoundKeyBegin; short Val;
286   SearchKey.GetKeyValue( 0, &FoundKeyBegin, Val );
287
288   EventEntry EE;
289   SearchKey.GetData( &EE, sizeof( EventEntry ) );
290   long FoundKeyEnd = FoundKeyBegin + EE.Span;
291
292   if ( FoundKeyEnd < NewPosition ) {
293     throw ProgramError( "Jfile::Seek",
294                         "Cannot Position after all data" );
295   }
296   CurPosition = NewPosition;
297   return CurPosition;
298 }
299
300 // This public write method checks the mode.
```



```
301 // If we are in Dynamic mode, the notwrite
302 // method is used
303 int Jfile::Write( void* Buf, int Cnt ) {
304   if ( JfileMode & BASE_DYNAMIC )
305     return NotWrite( Buf, Cnt );
306   else
307     return NormalWrite( Buf, Cnt );
308 }
309
310 int Jfile::NormalWrite( void* Buf, int Cnt ) {
311   // Construct an Event for this write
312   JournalEvent Evt;
313   Evt.Offset = CurPosition;
314   Evt.Size   = Cnt;
315   memcpy( Evt.Data, Buf, Cnt );
316
317   // Record the event
318   long Position = Evt.Record( JournalFD );
319
320   // Register the event
321   RegisterEvent( Evt, Position );
322
323   return Cnt;
324 }
325
326 /*
327   This method shows how to write to a file but
328   not show it.  For example if a file is evolving
329   and you want some user or process to have a static
330   view of it for backup or other purposes.
331
332   Strategy:
333     Seek to the appropriate position in the base file
334     Read the data about to be overwritten
335     Create an event for the overtyped data.
336     Append the event to the log
337     Register the event
338     Write the data to the BaseStream
339
340   Issues:
341     NotWrite must preserve the former EOF to prevent
342       the static reader from consuming stuff that was
343       written afterwards.
344
345     Note that WriteBaseStream does not change BaseFileSize
346       and since BaseFileSize is used to indicate when
347       the base stream reaches it's end, reading will
348       stop at that mark.
349 */
350
351 int Jfile::NotWrite( void* buf, int Cnt ) {
352   // Allocate a buffer to hold the overtype data
353   Bufr ReadBuf( Cnt );
354
355   // If the event occured within the file
356   // Manufacture an inverse event
357   if ( CurPosition < BaseFileSize ) {
358
359     // Go to the base location
360     SeekBaseStream( CurPosition );
```

```
361     // Will not read beyond old end of base stream
362     int GotCnt = ReadBaseStream( ReadBuf.Ptr(), Cnt );
363
364     // Manufacture the inverse write event
365     JournalEvent Evt;
366     Evt.Offset = CurPosition;
367     Evt.Size   = GotCnt;
368     memcpy( Evt.Data, ReadBuf.Ptr(), GotCnt );
369
370     // Record the event
371     long Position = Evt.Record( JournalID );
372
373     // Register the event
374     RegisterEvent( Evt, Position );
375 }
376 // Go back to the target location
377 // Note that SeekBaseStream will not seek beyond
378 // eof of the base stream.  This version is used
379 // to go to the proper logical location and write
380 // the data.
381 ForceSeekBaseStream( CurPosition );
382 // Write the actual data
383 WriteBaseStream( Buf, Cnt );
384 CurPosition += Cnt;
385
386 return Cnt;
387 }
```

```
/*
Program to create and use a Jfile

A Jfile is a demo file interface which uses
a Journal file in a variety of ways
  1) To simulate a standard file interface using a read only
     base file and an event log.
  2) To enable a non-seekable base file to be merged with
     an event log to another stream
  3) To demonstrate the use of an event log to emulate
     a seekable file
  4) To demonstrate the use of an event log to record
     inverse event and hold a static view of the base
     data while the base data evolves
*/

#include <jfile.h>
#include <cmdline2.h>
#include <iostream.h>
#include <fstream.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
  CmdLine2 CL;
  CL.Init( argv );

  if ( argc < 2 ) {
    cout << "jfile <options>" << endl;
    cout << "  bfs=source>     Base File Name" << endl;
    cout << "  jfs=source>     Journal File Name" << endl;
    cout << "  ofs=target>     Capture output to file" << endl;
    cout << "  read            Read Test" << endl;
    cout << "     seek=pos>    Seek to position before reading" << endl;
    cout << "     nobase       Do not open a base file" << endl;
    cout << "  write           Write some data" << endl;
    cout << "     str=string>  String to write" << endl;
    cout << "     seek=pos>    Write location" << endl;
    cout << "  baseseekable    Base Stream Seekable" << endl;
    cout << "  basedynamic     Mask changes to base file" << endl;
    cout << "  startcmd=string Run this file before starting" << endl;
    return 0;
  }

  if ( CL.FindName("") ) {
    CL.GoHome();
    cout << "Working Directory: " << CL.GetHome() << endl;
  } else
    cout << "Working Directory: " << CL.GetWhere() << endl;

  if ( CL.Find( "startcmd" ) ) {
    $buf StartCmd;
    CL.getline( StartCmd.Ptr(), StartCmd.size(), 0 );
    system( StartCmd.Ptr() );
  }

  $buf JournalFileName = "jrn";
  if ( CL.Find( "jfs" ) ) {
    $buf BaseFileName = "base";
    if ( CL.Find( "bfs" ) ) {
      CL.getline( BaseFileName.Ptr(), BaseFileName.size(), 0 );
    }
    int BaseSeekable = 0;

    try {
      // Share a Jfile instance
      Jfile Jf;

      if ( CL.Find( "baseseekable" ) )
        Jf.ToggleMode( BASE_SEEKABLE );

      if ( CL.Find( "basedynamic" ) )
        Jf.ToggleMode( BASE_DYNAMIC );

      Jf.Open( JournalFileName, BaseFileName );

      // Write a buffer of data to the string
      if ( CL.Find( "write" ) ) {

        $buf WriteString = "default_string";
        if ( CL.Find( "str=" ) )
          CL.getline( WriteString.Ptr(), WriteString.size(), 0 );

        // Jf.Open( JournalFileName, BaseFileName );

        long SeekPos = 0;
        if ( CL.Find( "seek" ) ) {
          CL >> SeekPos;
        }
        Jf.Seek( SeekPos );
        Jf.Write( WriteString.Ptr(), WriteString.length() );
        // try it again to test write past eof
        // Jf.Write( WriteString.Ptr(), WriteString.length() );
        // SeekPos = 0;
      }

      if ( CL.Find( "read" ) ) {
        int RedirectFD = -1;
        if ( CL.Find( "ofs" ) ) {
          $buf OutFileName;
          CL.getline( OutFileName.Ptr(), OutFileName.size(), 0 );
          RedirectFD = _creat( OutFileName, _S_IREAD | _S_IWRITE );
        }
        if ( CL.Find( "nobase" ) ) {
          BaseFileName = "";
        }

        Buff Buf( 100 );
        int GotCnt;

        long SeekPos = 0;
        if ( CL.Find( "seek" ) ) {
          CL >> SeekPos;
        }
        Jf.Seek( SeekPos );
```

```
121
122     do {
123        GotCnt = JF.Read( Buf.Ptr(), 100 );
124        if ( GotCnt <= 0 ) break;
125        if ( RedirectFD != -1 ) {
126           write( RedirectFD, Buf.Ptr(), GotCnt );
127        } else {
128           cout.write( Buf.Ptr(), GotCnt );
129           cout.flush();
130        }
131     } while ( 1 );
132     // JF.Close();
133     if ( RedirectFD != -1 ) close( RedirectFD );
134
135     JF.Close();
136
137  } catch ( Exception& EX ) {
138     char* ErrorString = EX.GetReportString();
139     return -1;
140  }
141  return 0;
}
```

```
 1 /*
 2   jrndelta.cpp
 3
 4   This file implements the methods which package
 5   an event journal as a delta file.
 6
 7   This method fills out the frame header for the
 8   next frame and inserts the appropriate data into
 9   the data buffer
10
11   Note:
12     This program does not attempt to maintain a signature
13     for the streams.  It is possible to maintain an xhash
14     term as in terac.
15
16     Briefly, use an n-byte (4) wide term and exclusive or
17     every quadword for the first pass.  Next to adjust the
18     signature for the file for each event, exclusive or
19     the written data divided into quadwords with the
20     signature for the earlier file.
21
22     Note that the offset of the event must be used to
23     ensure that the appropriate terms are xored as below
24
25   Base file:
26     abcdefgfjklmnopqrstuvwxyz
27     |   |   |   |   |   |   |
28
29   Event:
30       Hello dolly
31
32   Base term is:
33     abcd
34    ^efgh     ^ is exclusive-or
35    ^ijkl
36    ^mnop
37    ^qrst
38    ^uvwx
39    ^yz
40    ======
41    2F8C in hex (guessing only)
42
43   To adjust the signature for the event
44    2F8C
45   ^__H  (_) character is null
46   ^ello
47   ^ dol
48   ^ly
49   ======
50    9831 in Hex (guessing only)
51
52
53 */
54 #include <jrndelta.h>
55
56 JournalDelta::JournalDelta() : DataBuffer( MAXFRAMESIZE )
57 {
58   BaseStreamSize = -1;
59   CurPosition    = -1;
60 }
```



```
61 JournalDelta::~JournalDelta() {
62
63
64
65 int JournalDelta::ReadFrame( FrameHdr& FH, void* &DataPtr ) {
66
67   //
68   // This utility requires that a base file size have
69   // been set to enable frame constructions to properly
70   // handle the end of file conditions
71   //
72   if ( BaseStreamSize == -1 ) {
73     throw ProgramError( "JournalDelta::ReadFrame",
74                         "Size of base file must be set" );
75   }
76   // Normally we would install some meaningful data into
77   // the first frame.  But it is inconsequential to
78   // this example
79   if ( CurPosition == -1 ) {
80     FH.Key = FIRST;
81     FH.Len = sizeof( FF );
82     DataPtr = &FF;
83     // Advance to the first logical byte of the file
84     CurPosition = 0;
85     return FH.Len;
86   }
87
88   // If we have processed the LastFrame, then we have
89   // exhausted the frames.
90   // Just return EOF.
91   if ( FH.Key == LAST ) {
92     return EOF;
93   }
94   // Query the current position and construct a frame
95   EventEntry EE;
96   __int64 Stat = QueryLocation( CurPosition, EE );
97
98   // Return a data frame
99   if ( Stat == 0 ) {
100    // Go to the position having the data in the journal file
101    lseek( JournalFD, EE.Position, SEEK_SET );
102    // Copy the subject data from the journal file
103    // into the data buffer
104    int GotCnt = read( JournalFD, DataBuffer.Ptr(), EE.Span );
105    if ( GotCnt != EE.Span ) {
106      throw FileError( "JournalDelta::ReadFrame", JournalFileName );
107    }
108
109    // Now construct the frame header
110    FH.Key = DATA;
111    FH.Len = EE.Span;
112    // Set the pointer to the subject data
113    DataPtr   = DataBuffer.Ptr();
114
115    // Advance the current position by the size of
116    // the current frame
117    CurPosition += FH.Len;
118    return ( sizeof ( FH ) );
119  } else if ( Stat > 0 ) {
120    // Then we hit a matching section
```

```
121  // Stat contains the number of characters until
122  // the next entry
123  FB.Key = I_MATCH;
124  FB.Len = sizeof ( Matchframe );
125  NF.f1_beg = CurPosition;
126  NF.f2_beg = CurPosition;
127  NF.f2_end = CurPosition + Stat;
128
129  // Increment the size of the output stream
130  LF.f2_Len = NF.f2_end;
131
132  // Advance the current position to the next frame
133  CurPosition += Stat;
134
135  // Set the return pointer to the match frame
136  DataPtr = &NF;
137
138  return ( sizeof( FB ) );
139
140  } else {  // Stat == -1
141
142  if ( CurPosition >= BaseStreamSize ) {
143    // Last frame
144    // Signatures are not used in this mechanism
145    // Unless they are explicitly supported by
146    // reading the source file.  Note that it
147    // is possible to use a xhash style checksum
148    // to validate the files instead of a crc.
149    // For now this signature maintenance is omitted.
150    LF.f1_Sig = -1;
151    LF.f2_Sig = -1;
152    LF.f1_Len = BaseStreamSize;
153    // No skewing is possible so
154    // this was a static file
155    LF.Model  = STATIC;
156    // The new size is the maximum of the largest match
157    // or the greatest event stored
158    LF.f2_Len = max( NF.f2_end, GetLastEventPosition() );
159    DataPtr = &LF;
160    // Reset the header key key
161    FB.Key = LAST;
162    FB.Len = sizeof( LF );
163    return sizeof ( FB );
164  }
165  // There are no more frames until the end of the file
166  // Construct a match frame which spans to end of the base
167  // stream
168  FB.Key    = T_MATCH;
169  FB.Len    = sizeof ( Matchframe );
170  NF.f1_beg = CurPosition;
171  NF.f2_beg = CurPosition;
172  NF.f2_end = BaseStreamSize;
173
174  // Advance the current position to the next frame
175  CurPosition = BaseStreamSize;
176
177  // Set the return pointer to the match frame
178  DataPtr = &NF;
179
180  return ( sizeof( FB ) );
```

```
181 }
182
183 return sizeof( FB );
184 }
```

```
 1 #include <jrnsvc.h>
 2 #include <sys/stat.h>
 3
 4 JournalServiceBase::JournalServiceBase() {
 5   JournalFD = -1;
 6 }
 7
 8 JournalServiceBase::~JournalServiceBase() {
 9   if ( JournalFD != -1 )
10     JournalServiceBase::Close();
11 }
12
13 void JournalServiceBase::Open( char* JournalFile )
14 {
15   JournalFD = open( JournalFile, O_RDWR | O_BINARY );
16
17   if ( JournalFD == -1 ) {
18     JournalFD = open((const char *)JournalFile,
19           O_CREAT | O_TRUNC | O_RDWR | O_BINARY,
20           S_IREAD | S_IWRITE );
21   }
22   if ( JournalFD == -1 )
23     throw FileError( "JFile::Open", JournalFile );
24
25   JournalFileName = JournalFile;
26   JournalIxName   = JournalFile;
27
28   // Add a .i to the end of the journal name
29   char *Ext = strchr( JournalIxName, '.' );
30   if ( Ext ) strcpy( Ext, ".i" );
31   else JournalIxName += ".i";
32   int Stat;
33   try {
34     Stat = VT.Open( JournalIxName, VtFile::READ_WRITE );
35   } catch ( FileError& FE ) {
36     FE = FE;
37     Stat = VT_FAIL;
38   }
39   if ( Stat == VT_FAIL ) {
40     Stat = VT.Create( JournalIxName );
41     // Set up the key format for the tree
42     char kv[4];
43     memset( kv, 0, 4 );
44
45     kv[0] = Entryb::INT32;  //  Offset in base file
46     // Install the key template into the tree
47     VT.SetKeyVector( kv );
48
49     // Build an index for the journal
50     IndexJournalFile();
51   }
52   if ( Stat == VT_FAIL ) {
53     throw FileError( "JFile::Create", JournalIxName );
54   }
55 }
56
57 void JournalServiceBase::Close( )  {
58   //AddBaseEofMarker();
59   VT.Close();
60   close( JournalFD );
61   JournalFD = -1;
62 }
63 //
64 //  This method looks up the last event and returns
65 //  its trailing position.
66 //
67 long     JournalServiceBase::GetLastEventPosition() {
68   VT.AfterMaxEntry();
69   Entryb Ent( VT );
70   int Stat = VT.Backward( Ent );
71   if ( Stat == VT_FAIL ) {
72     return -1;
73   }
74   EventEntry EE;
75   // Calculate the last event position
76   // Load the starting position of the found key
77   long FoundKeyBegin; short Val;
78   Ent.GetKeyValue( 0, &FoundKeyBegin, Val );
79
80   Ent.GetData( EE, sizeof( EventEntry ) );
81
82   return (long)(EE.Span + FoundKeyBegin);
83 }
84 /*
85   This method queries a location and returns relevant
86   information.
87
88   1)  If the location is within an event, the EE
89       structure is filled and zero is returned.
90
91   2)  If the location is not in an event, the
92       number of characters until the next event
93       are returned.
94
95   3)  If there are no subsequent events, then -1 (EOF)
96       is returned.
97
98   Note:
99     Certain applications may require tracking of the base
100    file size.  It is possible to use this object without
101    ever establishing the size when:
102      1)  The event journal does not include a size marker.
103      2)  When the application does not provide one automatically.
104
105
106 */
107 __int64 JournalServiceBase::QueryLocation( __int64 Location, EventEntry& EE )
108 {
109
110   // Construct a search entry
111   Entryb SearchKey( VT );
112   SearchKey.AddKeyElement( (long) Location );
113
114   // Go to the exact key or next larger or end of tree
115   int Stat = VT.FindClosest( SearchKey );
116   if ( Stat == VT_FAIL )
117     return -1;  // No Events in journal
118
119   SearchKey.printOn ( cout );
120
```

3

```
121  // Load the starting position of the found key
122  long FoundKeyBegin; short Val;
123  SearchKey.GetKeyValue( 0, &FoundKeyBegin, Val );
124
125  // If we found an exact match then return
126  // the entry information
127  if ( FoundKeyBegin == Location ) {
128    SearchKey.GetData( &EE, sizeof( EventEntry ) );
129    return 0;  // Event Found
130  }
131
132  // Otherwise the found event is beyond the location
133  // Remember the space between the location and
134  // the found event
135  __int64 DistanceToNextEvent = FoundKeyBegin - Location;
136
137  // Now Back up by an event
138  // If this is the last entry in the tree
139  // We might be within its span
140  if ( FoundKeyBegin > Location ) {
141    Stat = VT.Backward( SearchKey );
142    if ( Stat == VT_FAIL ) {
143      // No earlier event tell caller how far to next one
144      return DistanceToNextEvent;
145    }
146  }
147
148  // So we should be at the event just before the location
149  // Check to see if Location is within the event
150  SearchKey.printOn ( cout );
151
152  // Load the starting position of the found key
153  // long FoundKeyBegin; short Val;
154  SearchKey.GetKeyValue( 0, &FoundKeyBegin, Val );
155
156  EventEntry FEE;
157  SearchKey.GetData( &FEE, sizeof( EventEntry ) );
158  long FoundKeyEnd = FoundKeyBegin + FEE.Span;
159
160  // If Location is beyond the earlier event,
161  // Then location is not within an event
162  // Report distance to next event
163  if ( Location >= FoundKeyEnd ) {
164    if ( DistanceToNextEvent < 0 ) return -1;
165    else return DistanceToNextEvent;
166  }
167
168
169  // Otherwise Location resides within an event
170  // Build a EventEntry to notify the caller
171  // Of both the offset and span and offset
172  // of the info available in the journal file
173
174  //  0123456789012
175  //  1++++ 2++ 3++
176  //
177  //  Location = 3
178  //  SearchKey = 0_pos,4_span
179  //  JournalFileSpan   = How much span after location
180  //  JournalFileOffset = OldJrnPosition + (OldSpan - JournalFileSpan)
```


4

```
181    EE.Span     = FEE.Span - (Location - FoundKeyBegin);
182    EE.Position = FEE.Position + ( FEE.Span - EE.Span );
183    return 0;
184  }
185
186
187
188  #include <fstream.h>
189  #include <stdlib.h>
190
191  /*
192    This method takes a journal file and creates an
193    index.  The index is linearly searchable.
194    The index can be used to sequentially read segments
195    of the final stream.
196  */
197  void JournalServiceBase::IndexJournalFile( ) {
198
199    // Create an event object to read into
200    JournalEvent Evt;
201
202    // Open the Journal
203    ifstream IS( JournalFileName, ios::binary );
204
205    // Cycle through the journal
206    do {
207      // Point to first character of data in the file
208      // Just beyond the event header
209      long Position = IS.tellg() + sizeof(long) + sizeof(short);
210
211      Evt.Load( IS );
212
213      // Stop at eof
214      // Note that we can stop at any event to build as
215      // of that point in time
216      if ( !IS ) break;
217      Evt.printOn( cout );
218
219      RegisterEvent( Evt, Position );
220
221    } while( IS ); // while getting events
222  }
223
224
225  void JournalServiceBase::RegisterEvent( JournalEvent& Evt, long Position ) {
226    // Note that the disclosure document recites using a pair
227    // of entries, one marking the beginning and one marking
228    // the end.  Since this Tree embodyment permits
229    // efficient revision of tree data values, it is more efficient
230    // to revise the segment length in the data area of the node.
231    // This way a single key contains all the information needed
232    // to represent an event.  Benefit Summary:
233    // 1)  Half as many keys in the tree
234    // 2)  Less deletions
235
236    // The stream is positioned beyond the event at the beginning
237    EventEntry EE;
238    EE.Position = Position;
239
240    EntryD EventKey( VT );
```

```
241   // The starting entry uses the offset as its beginning
242   EventKey.AddKeyElement( Evt.Offset );
243   EE.Span = Evt.Size;
244   EventKey.SetData( &EE, sizeof( EE ) );
245
246
247   /* Before we can insert the key, we have to make sure
248      there are no overlapping events before we can insert the
249      current event.
250
251      Example 1:
252                1111111111222222222233
253      01234567890123456789012345678901
254      1++++___3++++__2+++__4++___  //  Current Events
255        5++++++++++++++          //  New Event
256
257      In this example event 4 has overwritten part of event1
258      all of event 3 and the first part of event 2.
259
260      Event 1 has to be reduced to only 2 characters.
261      Event 3 must be deleted entirely
262      Event 2 must be reduced by the first two characters
263      Example 2:
264                1111111111222222222233
265      01234567890123456789012345678901
266      1++++___3++++__2+++__4++___  //  Current Events
267        5+                         //  New Event
268
269      In this example event 5 is contained within event 1
270      1)  Character 4 of event 1 must become an independent event
271      2)  Event 5 must be an event
272      3)  The leading part of event 1 must be shortened to 2 chars
273
274      Example 3:
275                1111111111222222222233
276      01234567890123456789012345678901
277      1++++___3++++__2+++__4++___  //  Current Events
278        5+++++++++                 //  New Event
279
280      In this example event 5 is spans part of 1 and 3
281      1)  Event 3 must be replaced with an event of 2 chars at offset 11
282      2)  Event 1 should be shortened to 2 characters
283
284   */
285   // Locate the first event after the current
286   // or possibly the
287   EntryB SearchKey( VT );
288
289   // Position the tree trailing part of the current event
290   long  EventBeginPoint = Evt.Offset;
291   long  EventEndPoint   = Evt.Offset + Evt.Size;
292   short EventSpan = Evt.Size;
293   SearchKey.AddKeyElement( EventEndPoint );
294
295   // Go to the exact key or next larger or end of tree
296   int Stat = VT.FindClosest( SearchKey );
297
298   // If a key no key retrieved, tree is empty
299   if ( Stat == VT_FAIL ) {
300     Stat = VT.Add( EventKey );
```



```
301     return;
302   }
303
304   while ( Stat == VT_SUCCESS ) {
305     SearchKey.printOn ( cout );
306
307     // Load the starting position of the found key
308     long FoundKeyBegin; short Val;
309     SearchKey.GetKeyValue( 0, &FoundKeyBegin, Val );
310
311     EventEntry FEE;
312     SearchKey.GetData( &FEE, sizeof( EventEntry ) );
313     long FoundKeyEnd = FoundKeyBegin + FEE.Span;
314
315     // If the element is beyond the event go backward
316     if ( EventEndPoint <= FoundKeyBegin ) {
317       Stat = VT.Backward( SearchKey );
318       continue;
319     }
320
321     //  Is the trailing marker within an event
322     if ( EventEndPoint < FoundKeyEnd ) {
323       // Create a residual event marker and insert it
324       // Note:  at this time both the original event marker
325       //   and this new one cover the span.  We'll delete
326       //   the other one later.
327       EntryB NewKey( VT );
328       long NewBeginPoint = EventEndPoint;
329       long NewSpan       = FoundKeyEnd - EventEndPoint;
330       NewKey.AddKeyElement( NewBeginPoint );
331
332       // The data will be different
333       EventEntry EvNew;
334       // Span will adjust downward
335       EvNew.Span = NewSpan;
336       // Position in the journal file will adjust upward
337       EvNew.Position = FEE.Position + EventEndPoint - FoundKeyBegin;
338
339       NewKey.SetData( &EvNew, sizeof( EvNew ) );
340       // NewKey.data_field = NewSpan;
341       VT.Add( NewKey );
342       // Go back to the search key
343       Stat = VT.FindExact( SearchKey );
344     }
345
346     // It is possible to revise the earlier key if
347     // it is the same as the former.  For now
348     // simply delete the former
349
350     // If this is a wholly contained event, delete it
351     if ( EventBeginPoint <= FoundKeyBegin ) {
352         // && EventEndPoint > FoundKeyEnd ) { // Wholly contained
353         Stat = VT.Delete( SearchKey );
354         // check tree position here??
355         Stat = VT.Backward( SearchKey );
356         continue;
357     }
358
359     // Check to see if the new key begins after the old one ends
360     // But there must be overlap
```

```
361     // ( EventBeginPoint > FoundKeyBegin
362           && EventBeginPoint < FoundKeyEnd  // Must overlap
363        )
364        {
365           // Shorten the old key
366           FEE.Span = EventBeginPoint - FoundKeyBegin;
367           // Shorten the span of the key where we landed
368           VI.UpdateData( EFEE, sizeof( FEE ) );
369           // No earlier keys will reference this one
370        }
371        break;
372     } // encountering events in index
373     slot = VI.Add( EventKey );
374  }
375 }
```

```
 1 /*
 2   This module contains miscellaneous service routines for
 3   event merging logic.
 4 */
 5 #include <fstream.h>
 6 #include <jrnsvc.h>
 7 //#include "jrnevt.h"
 8 //#include <vtree.h>
 9
10 void BuildIndex( char* JournalName ){
11   JournalServiceBase JS;
12   JS.Open( JournalName );
13   JS.Close();
14 }
15
16 // Logname provides the name for the output file
17 // EventCnt is the number of events to be generated
18 // SrcFileSize is the highest upper bound of an event
19
20 void BuildRandomEventLog( char* LogName,
21                           int   EventCnt = 10,
22                           long  SrcFileSz = 10240,
23                           int   MaxEventSz = 1024  // Largest permitted event
24                          ) {
25
26
27   // Number of entries to generate
28   srand(0);
29   cout << "LogName:     " << LogName << endl;
30   cout << "SrcFileSz:   " << SrcFileSz << endl;
31   cout << "EventCnt:    " << EventCnt << endl;
32   cout << "MaxEventSz:  " << MaxEventSz << endl;
33
34   // Create the journal file
35   ofstream OS( LogName, ios::binary );
36   JournalEvent Evt;
37
38   for ( int i = 0; i < EventCnt; i++) {
39
40     // Reset the Event
41     Evt.Clear();
42
43     // Synthesize an event
44     // Make up an offset
45     int Offset = rand( ) % SrcFileSz;
46
47     // Make up a size for the event
48     int EvtSize = rand( ) % MaxEventSz;
49
50     // Note that the event may extend the file so
51     // increment the source file size if necessary
52     SrcFileSz = ( SrcFileSz > EvtSize + Offset )
53                       ? SrcFileSz : EvtSize + Offset;
54
55     // Now make up some text data for the event
56     // Pick a letter of the alphabet
57     int RandomLetterIx = rand( ) % 26;
58     memset( Evt.Data, RandomLetterIx + 'A', EvtSize );
59
60     // Set the rest of the event
```



```
61     Evt.Offset = Offset;
62     Evt.Size = (short)EvtSize;
63
64     Evt.Record( OS );
65   }
66 }
67
68 void ListEventLog( char* LogName ) {
69   ifstream IS( LogName, ios::binary );
70   JournalEvent Evt;
71   cout << "Printing Journal" << endl;
72   do {
73     Evt.Load( IS );
74     if ( !IS ) break;
75     Evt.printOn( cout );
76   } while ( 1 );
77 }
78
79
80 /*
81   Quick and dirty interface to record a manual
82   event journal to create test cases
83 */
84
85 void BuildManualEventLog( char* LogName ) {
86
87   cout << "LogName:     " << LogName << endl;
88
89   // Create the journal file
90   ofstream OS( LogName, ios::binary );
91   JournalEvent Evt;
92
93   do {
94     // Reset the event
95     Evt.Clear();
96     cout << "Enter Offset:  (-1 to end)" << ends;
97     cin >> Evt.Offset;
98     if ( Evt.Offset == -1 ) break;
99
100    cout << "Enter String:              " << ends;
101    cin >> Evt.Data;
102    Evt.Size = strlen( Evt.Data );
103
104    Evt.printOn( cout );
105    Evt.Record( OS );
106  } while ( 1 );
107 }
108
109 /*
110   This procedure lists the contents of an index
111 */
112 void ListIndex( char* IxName ) {
113   Vtree VT;
114   VT.Open( IxName, VtFile::READ_WRITE );
115
116   Entryb E( VT );
117   cout << "Listing Index:  " << IxName << endl;
118   VT.BeforeMinEntry();
119   while ( VT.Forward() == VT_SUCCESS ) {
120     VT.GetCurr( E );
```

000028

```
121     E.printOn( cout );
122     EventEntry EE;
123     E.GetData( &EE, sizeof( EE) );
124     cout << "Journal Data: ";
125     EE.printOn( cout );
126   }
127 }
```

C:\DELTA\PATENT\LOGFILES\SRC\MISC.CPP

Mon Sep 30 21:47:23 1996

```
/*
   this file demonstrates a means of building
   an index from a logfile of events.
*/
#include <vtree.h>
#include <cmdline2.h>

void BuildRandomEventLog( char*, int, long, int );
void BuildManualEventLog( char* );
void ListEventLog( char* LogName );
void ListIndex( char* );
void BuildIndex( char* );

int main(int argc, char *argv[])
{
  cmdline2 CL;
  CL.Init( argv );

  if ( argc < 2 ) {
    cout << "usage:tm <options>" << endl;
    cout << "    [name=<test.idx>" << endl;
    cout << "    [f=<source>      Use this as source" << endl;
    cout << "    base=<source>    Use file as base text source" << endl;
    cout << "    o=<target>       Put results here" << endl;
    cout << "    list             Print out" << endl;
    cout << "    eventlog         List the event log" << endl;
    cout << "    index            List the build index" << endl;
    cout << "    home=homedir>    Use this as working directory" << endl;
    cout << "    make             Construct Something" << endl;
    cout << "    randomeventlog   Construct an event log" << endl;
    cout << "       eventcnt<10>  Number of events to put in log" << endl;
    cout << "       sourcesize<10240> Size of source file" << endl;
    cout << "       maxevent<1024> Largest event" << endl;
    cout << "    index            Build an index for a journal" << endl;
    cout << "    log              Manually build an event log" << endl;
    cout << "    yack       Echo key content before action" << endl;
    //cout << "    revert     Revert index to original state" << endl;
    return 0;
  }
  if ( CL.FindBone() ) {
    CL.GetName();
    cout << "Working Directory:  " << CL.GetName() << endl;
  } else
    cout << "Working Directory:  " << CL.GetHere() << endl;

  Sbuf IndexName = "test.i";
  if ( CL.Find( "iname" ) ) {
    CL.GetLine( IndexName.Ptr(), IndexName.size(), 0 );
  }
  Sbuf SourceName = "source";
  if ( CL.Find( "if" ) ) {
    CL.GetLine( SourceName.Ptr(), SourceName.size(), 0 );
  }
  Sbuf DestName = "target";
  if ( CL.Find( "of" ) ) {
    CL.GetLine( DestName.Ptr(), DestName.size(), 0 );
  }
  Sbuf LogFileName = "logfile";
  if ( CL.Find( "logfile" ) ) {
    CL.GetLine( LogFileName.Ptr(), LogFileName.size(), 0 );
    int Yack = 0;
    if ( CL.Find( "yack" ) ) Yack = 1;

    if ( CL.Find( "make" ) ) {
      // Synthesize an event log
      if ( CL.Find( "randomeventlog" ) ) {
        cout << "Building Event Log" << endl;
        int EventCnt = 10;
        if ( CL.Find( "eventcnt" ) ) CL >> EventCnt;
        long SourceSize = 10240;
        if ( CL.Find( "sourcesz" ) ) CL >> SourceSize;
        int MaxEventSize = 1024;
        if ( CL.Find( "maxevent" ) ) CL >> MaxEventSize;
        BuildRandomEventLog( DestName, EventCnt, SourceSize, MaxEventSize );
        SourceName = DestName;
      }
      else if ( CL.Find( "log" ) ) {
        cout << "Creating Manual Event Log" << endl;
        BuildManualEventLog( DestName );
        SourceName = DestName;
      }
      else if ( CL.Find( "index" ) ) {
        cout << "Building Index from Event Log" << endl;
        BuildIndex( SourceName );
        SourceName = DestName;
        SourceName += ".i";
      }

    if ( CL.Find( "list" ) ) {
      ListEventLog( SourceName );
    } catch ( Exception EX ) {
      EX.Report();
    }
    ListIndex( SourceName );
  }
  return 0;
}
```

```
 1 /*
 2   jfile.hpp is the prototype for jfile
 3 */
 4 #ifndef _JFILE_H_
 5 #define _JFILE_H_
 6
 7 #include <jrnsvc.h>
 8 //#include <vtree.h>
 9 //#include <buf.h>
10 //#include "../jrnsvt.h"
11
12 // These parameters are installed into JfileMode.  They
13 // set particular behaviors of the jfile.  The purpose
14 // of these parameters is to demonstrate multiple behaviors
15 // of Jfile without creating a bunch of independant modules.
16 // The operating modes are briefly described below:
17
18 // The base file is seekable. This mode permits random
19 // position access via the seek command.  The read and
20 // write can be used to replace or retrieve data at that
21 // location.
22 // The seek command can be used to position the Jfile
23 // to any location within the file's domain and the data
24 // there can be read
25
26 // A non-seekable base is to be used when the base stream
27 // comes from a tape or other device which cannot seek
28 // backwards and do not know the length of the base stream.
29 // The length of the base stream is only known when EOF
30 // is encountered.
31 #define  BASE_SEEKABLE        ( 1 )
32 // A static base (default) means that the base file is not permitted
33 // to change.  Changes are recorded into the event journal only.
34 // Reading from a static jfile dynamically assembles data
35 // from the base file and the event journal to service the
36 // read request.
37 //
38 // BASE_DYNAMIC permits changes to be recorded to the base file.
39 // But the inverse event is recorded to the event jourel.
40 // The Read interface uses the inverse event journal to mask out
41 // changes made to the base.
42 //
43 // Note that BASE_DYNAMIC is only useful with a Seekable base.
44 #define  BASE_DYNAMIC         ( 1<<1)  // Base does not change
45
46 class Jfile : public JournalServiceBase {
47
48 protected:
49   /*
50   Vtree VT;
51
52   Sbuf     JournalFileName;
53   Sbuf     JournalIxName;
54
55   int      JournalFD;
56   */
57   __int64  CurPosition;
58   __int64  CurLength;
59
60   __int64  BaseStreamPosition;  // Current offset in base stream
61   int      JfileMode;           // Flag to determine if base can seek
62
63 protected:
64   //void     IndexJournalFile( );
65   //void     RegisterEvent( JournalEvent& JE, long Position );
66   //__int64  QueryLocation( __int64 Location, EventEntry& EE );
67
68   // Virtualize the interface to the base file so this
69   // descendants of this class can be used to demonstrate
70   // other types of base file activities.  IE tape, CDR
71   // and etc.
72
73   // Note these variables may not be used by the derived class
74   Sbuf     BaseFileName;
75   int      BaseFD;
76   long     BaseFileSize;
77
78   // Use this method to capture discarded buffers in
79   // a derived class.  In the context of a versioning product
80   // these chunks would be used as reverse match frames
81   virtual void  DiscardBaseStreamSegment( void*, int )();
82   virtual void  OpenBaseStream( char* BaseName );
83   virtual int   ReadBaseStream( void* Buf, int Cnt );
84   virtual void  CloseBaseStream();
85   virtual void  SeekBaseStream( long NewPosition );
86
87   // When in dynamic mode it is sometimes necessary to
88   // seek the logical end of file maintained by SeekBaseStream
89   virtual void ForceSeekBaseStream( long Position );
90
91   // This method supports nowrite mode
92   virtual int   WriteBaseStream( void* Buf, int Cnt );
93
94   // This method performs a write to the event Journal
95   // and causes the logical result to be displayed
96   // Base file appears modified but is not
97   int      NormalWrite( void* Buf, int Cnt );
98
99   // This method causes the base file to be modified
100  // But reading the Jfile will not show the change.
101  int      NotWrite( void* Buf, int Cnt );
102
103 public:
104   Jfile();
105   ~Jfile()();
106
107   void     Open( char* JournalFile, char* BaseFile = NULL );
108   void     Create()();
109   int      Read( void* Buf, int Cnt );
110   int      Write( void* Buf, int Cnt );
111   __int64  Seek( __int64 Offset );
112   void     Close();
113
114   void     ToggleMode( int Mask ) {
115     JfileMode ^= Mask;
116     // To use dynamic mode, the base must be seekable
117     if ( JfileMode & BASE_DYNAMIC )
118       JfileMode |= BASE_SEEKABLE;
119   }
120   int      GetMode() { return JfileMode; }
```

Mon Sep 30 21:30:52 1996
C:\DELTAVPATENT\LOGFILE\MCVJFILE.#
_squibs
(21);;
122
123 endif _JFILE_N_
3
4
3651159
3650809

```
1 /*
2  The purpose of this class is to translate an event
3  journal into a delta stream.
4
5  A delta stream provides alternating sequence of frames indicating
6  matching and mismatched text in a source and target file.
7
8  See associated C++ API for further description of data
9  files.
10
11  This class uses an index to collate events and queries
12  the index in response to a request frame.
13 */
14 #ifndef _JRNDELTA_H_
15 #define _JRNDELTA_H_
16
17 #include <jrnsvc.h>
18 #include <terec.h>
19
20 class JournalDelta : public JournalServiceBase {
21 protected:
22   // Standard frames to return data
23   FrameHdr  HH;
24   MatchFrame MF;
25   LastFrame  LF;
26   FirstFrame FF;
27   // Buffer to hold miss frame data
28   Buf       DataBuffer;
29   long      BaseStreamSize;
30
31   // Counter to keep track of the current position
32   long      CurPosition;
33
34 public:
35   JournalDelta();
36   ~JournalDelta();
37
38   // Manually set the size of the base file
39   void SetBaseStreamSize( long Size ) {
40     BaseStreamSize = Size;
41   }
42   long GetBaseStreamSize() { return BaseStreamSize; }
43
44   // Retrieves the next logical frame
45   int ReadFrame( FrameHdr& FH, void* pData );
46 };
47
48 #endif
```

```
 1
 2 #ifndef _JRNSVC_H_
 3 #define _JRNSVC_H_
 4
 5 #include <vtree.h>
 6
 7 #include <iostream.h>
 8 #include <iomanip.h>
 9 #include <memory.h>
10
11 // This structure is stored in the tree
12 // and holds relevant information about
13 // the event in the index
14 struct EventEntry {
15   long Span;
16   long Position;
17   EventEntry() {
18     Span=0;
19     Position = -1;
20   }
21   ostream& printOn( ostream& OS ) {
22     OS << setw( 8 ) << "Span:" << setw( 10 ) << Span;
23     OS << setw( 8 )  << "Pos:"  << setw( 10 ) << Position << endl;
24     return OS;
25   }
26 };
27
28 /*
29   This particular event is a sample.  All events should
30   include at least the fields represented in this event
31   or the equivalent
32 */
33 #define MAXEVENTSIZE ( 1024)
34 #define MAXPRINTEVENT ( 30 )
35 struct JournalEvent {
36   long  Offset;        // Position in source file
37   short Size;          // How much of data buffer is used
38   char  Data[MAXEVENTSIZE];   // Space for the event data
39
40   JournalEvent() {
41     Clear();
42   };
43   // Initialize the event
44   void Clear() {
45     Offset = -1;
46     Size = 0;
47     memset( Data, 0, MAXEVENTSIZE );
48   }
49
50   // Record the event into a stream
51   ostream& Record( ostream& OS ) {
52     OS.write( (char*) &Offset, sizeof( long ) );
53     OS.write( (char*) &Size, sizeof( short ) );
54     OS.write( Data, Size );
55     return OS;
56   }
57
58   // Record this event in a file
59   // Return the data position
60   long Record( int FD ) {
61     long EvtPosition = lseek( FD, 0, SEEK_END );
62     write( FD, &Offset, sizeof( long ) + sizeof( short ) + Size );
63     return EvtPosition + sizeof( long ) + sizeof( short );
64   }
65   // Read an event from a stream
66   istream& Load( istream& IS ) {
67     Clear();
68     IS.read( (char*)&Offset, sizeof( long ) );
69     IS.read( (char*)&Size, sizeof( short ) );
70     IS.read( Data, Size );
71     return IS;
72   }
73   // Print it out
74   ostream& printOn( ostream& OS ) {
75     OS << setw( 10 ) << "Offset:" << setw( 10 ) << Offset;
76     OS << setw( 8 )  << "Size:"   << setw( 10 ) << Size << "  ";
77     // Abreviate the print string for large ones
78     int PrtSize = ( Size < MAXPRINTEVENT ) ? Size : MAXPRINTEVENT;
79     OS.write( Data, PrtSize );
80     if ( Size != PrtSize ) OS << "...";
81     OS << endl;
82     return OS;
83   }
84 };
85
86 /*
87   Several methods and data are common to classes
88   which use Journals.  These common methods
89   are collected into this class.
90
91 */
92
93 class JournalServiceBase {
94 protected:
95
96   Vtree VT;
97
98   Sbuf      JournalFileName;
99   Sbuf      JournalIxName;
100
101   int       JournalFD;
102
103   // It is sometimes necessary to know the working length
104   // of the base file.  This working length may be the result
105   // of either the base file position, or some event which
106   // extended the file.
107   //long      BaseEofPosition;
108
109 protected:
110   void      IndexJournalFile(  );
111   void      RegisterEvent( JournalEvent& JE, long Position );
112   __int64   QueryLocation( __int64 Location, EventEntry& EE );
113
114   // Internal services to track the length of the base file
115   //void DropBaseEofMarker();  // Removes marker & sets BaseEofPos on open
116   //void AddBaseEofMarker();   // Adds marker at close
117
118 public:
119   JournalServiceBase();
120   ~JournalServiceBase();
```

000034
Tue Oct 01 07:46:44 1996
C:\DELTA\PATENT\LOG\FILE\INCL\JENSVC.H
_squiba
121 void Open(char* FileName);
122 void Close();
123 long GetLastEventPosition();
124
125
126 //void SetBaseEOFPosition(long Position) {
127 // BaseEofPosition = Position;
128 //}
129 };
130
131 #endif _JENSVC_H_

```
squibm                                C:\TMP\VI\BUCKETB.H                                Wed Jul 05 02:05:40 1995
```

```
 1 ////////////////////////////////////////////////////////////////////
 2 // bucketb.h: Cache bucket base class definition.
 3 // This has the data independent code for buckets.
 4 //                      * * *
 5 // NOTE: This file is slightly modified from what's in the
 6 // book. I've added a data_size field. This is used by the
 7 // Vnode cache. For the fixed-order-node (Knode) cache,
 8 // you can ignore this field.
 9 //                      * * *
10 // Copyright(c) 1993 Azarua Software. All rights reserved.
11 ////////////////////////////////////////////////////////////////////
12 #ifndef H_BUCKETB
13 #define H_BUCKETB
14
15 class Cacheb; class Coptrb; class Fmgr; // forward decl's
16
17 class Bucketb {
18 protected:
19   long addr;      // Location of bucket's data in the file
20   int refcnt;     // How many coptrs are pointing to this bucket
21   int data_size;  // Size of bucket's data << NEW FEATURE
22   char dirty;     // True when data doesn't match what's in file
23   Bucketb *prev;  // Pointers to adjacent buckets in the cache
24   Bucketb *next;  // in most-recently-acquired order.
25   void MakeNull() { addr = 0; refcnt = 0; dirty = 0; }
26 public:
27   Bucketb()           { } // Other classes initialize buckets
28   void Lock()         { ++refcnt; }
29   void Unlock()       { --refcnt; }
30   int IsLocked()      { return refcnt; }
31   void SetDirty()     { dirty = 1; }
32   int IsNull()        { return addr == 0; }
33   void Flush(Fmgr &f) { if (addr && dirty) Store(f); }
34   int Datasize()      { return data_size; } // NEW FEATURE
35   virtual void Fetch(Fmgr &f) = 0; // Depends on bucket data
36   virtual void Store(Fmgr &f) = 0; // Depends on bucket data
37   friend class Cacheb;
38   friend class Coptrb;
39   //C700 ERROR: redefining default parameter 2
40   //C700:  friend void Report(Cacheb &c, int full_report=0);
41   friend void Report(Cacheb &c, int full_report);
42 };
43
44 #endif
```

```
// character buffer class - a buffer with bounds checking
#include "buf.h"

#include <stdlib.h>
#include <string.h>

// #define DEBUG 1

//////////////////////////////////////////////////////////////////////
void* memset( Buf &B, int c, size_t n)
{
    if( n > B.size() )
        {
            B.Throw( ERR_BUF_BOUNDS, "memset trying to set beyond end of buffer" );
        return NULL;
        }
    return ::memset( (void*)B.ptr, c, n) );
}

//////////////////////////////////////////////////////////////////////
void* memcpy( Buf &B, const void *src, size_t n)
{
    if( n > B.size() )
        {
            B.Throw( ERR_BUF_BOUNDS, "Attempt to memcpy past buffer end" );
        return NULL;
        }
    return( ::memcpy( (void*)B.ptr, src, n) );
}

//////////////////////////////////////////////////////////////////////
// Default constructor for Buf class
//
Buf::Buf( void )
{
    bufsize=0;
    ptr = NULL;
}

//////////////////////////////////////////////////////////////////////
//  The copy constructor

Buf::Buf( const Buf& B )
{
    bufsize=B.size();
    if(!B.ptr) {
        ptr=NULL;
        return;
    }
    // allocate memory for buffer
    if ((ptr = new char[ bufsize] ) == NULL ) {
        Throw( ERR_BUF_ALLOC, "Allocation error in Buf::Buf( const Buf& B )" );
        return;
    }
    memcpy( (void*)ptr, B.ptr, bufsize);
}

//////////////////////////////////////////////////////////////////////
Buf::Buf( unsigned size )
{
    // allocate memory for buffer
    if ((ptr = new char[ bufsize] ) == NULL ) {
        Throw( ERR_BUF_ALLOC, "Allocation error in Buf::Buf( unsigned size )" );
        return;
    }
    if(ptr)
        memset( (void*)ptr,0,bufsize);
}

//////////////////////////////////////////////////////////////////////
// constructor - set a buffer giving a char string as an initializer

Buf::Buf( const char* ptr )
{
    int lenstrlen(ptr);
    bufsize=len+1;

    // allocate memory for buffer
    if ((ptr = new char[ bufsize] ) == NULL ) {
        Throw( ERR_BUF_ALLOC, "Allocation error in Buf::Buf( const char* ptr )" );
        return;
    }

    if(ptr)
        strcpy( ptr, ptr);
}

//////////////////////////////////////////////////////////////////////
// destructor

Buf::~Buf(void)
{
    bufsize=0;
    delete[] ptr;
    ptr=NULL;
}

//////////////////////////////////////////////////////////////////////
// realloc the buffer to a different size

int Buf::resize( unsigned NewSize )
{

    // Allocate a new buffer

    char* NewPtr = new char[ NewSize ];

    if ( NewPtr == NULL ) {
        Throw( ERR_BUF_ALLOC, "Allocation error in Buf::resize( unsigned NewSize )
    " );
        return( -1 );
    }

    // Initialize the contents of the newly allocated buffer
    memset( (void*)NewPtr, 0, NewSize );

_squibs ____________ C:\TMP\VF1\BUF.CPP ____________ Mon Sep 30 21:33:18 1996_
```

```
120   // Duplicate the contents of the current buffer
121   memcpy( (void*)NewPtr , ptr, bufsize );
122
123   // Delete the old buffer
124   delete[] ptr;
125
126   // Switch ptr to the newly allocated buffer
127   ptr = NewPtr;
128
129   // Reset the buffer size
130   bufsize = NewSize;
131   return( 0 );
132
133 )
134
135
136 ////////////////////////////////////////////////////////////////////////
137 // return the address of one character within the buffer
138 // -does bounds checking
139
140 char& Buf::operator()(unsigned i)
141 {
142   if( i>bufsize)
143     {
144       throw( ERR_BUF_BOUNDS, "Request outside buffer area: char& Buf::operator()
    (unsigned i) " );
145       return ptr[0];
146     }
147   return( ptr[i] );
148 }
149
150 ////////////////////////////////////////////////////////////////////////
151 // return the address of one character within the buffer
152 // -does bounds checking
153
154 char& Buf::operator[](int i)
155 {
156   if ( i < 0 ) {
157     Throw( ERR_BUF_BOUNDS, "Negative request for bugger data char& Buf::operator
    [](int i)" );
158     return ptr[0];
159   }
160   return Buf::operator[]( (unsigned)i );
161 }
162
163 ////////////////////////////////////////////////////////////////////////
164 // return the buffer size
165
166 unsigned Buf::size(void) const
167 {
168   return(bufsize);
169 }
170
171 ////////////////////////////////////////////////////////////////////////
172 //  What is the length of the string in buf
173
174 unsigned Buf::length(void) const
175 {
176   return(strlen( ptr ));
177 }
```



```
178
179 ////////////////////////////////////////////////////////////////////////
180 // Reset the contents of the buf to null
181
182 void Buf::clear(void) const
183 {
184   memset( (void*)ptr,0, bufsize);
185 }
186
187 ////////////////////////////////////////////////////////////////////////
188
189 ostream& operator << ( ostream& strm, const Buf *B)
190   {
191     B->printOn( strm );
192     return (strm);
193   };
194
195 ////////////////////////////////////////////////////////////////////////
196
197 ostream& operator << (ostream& strm, const Buf &B)
198   {
199     B.printOn( strm );
200     return (strm);
201   };
202
203 /*
204   Default throw function for the class
205 */
206 // Set up a default error handler
207 int Buf::Throw( int ErrCode, char* AuxText ) {
208
209   switch ( ErrCode ) {
210    case ERR_BUF_ALLOC:
211      cerr << "Memory allocation failed" << endl;
212      abort();
213      break;
214    case ERR_BUF_BOUNDS:
215      cerr << "Attempt to access outside allocated buffer" << endl;
216      abort();
217      break;
218    default:
219      // Return a not found marker
220      // error must belong to someone else
221      return 1;
222      // break;
223   }
224   return 0;
225 }
226
227 /*
228   Bufr Class definition
229
230 */
231 #include <string.h>
232
233 ////////////////////////////////////////////////////////////////////////
234 // overloaded operator for setting buffer equal to another buffer
235 // -does bounds checking
236
237 Bufr& Bufr::operator=(const Bufr& S)
```

```
238 {
239   unsigned int tsize=S.size();
240   if( tsize > bufsize)
241     if( resize(tsize) != 0 )
242       {
243         //Log( ER_NOMEM, "Unable to resize Bufr" );
244         return *this;
245       }
246   memcpy( ptr, S.ptr, tsize );
247   return *this;
248 }
249
250 ////////////////////////////////////////////////////////////////////////
251 // overloaded operator for setting buffer equal to the contents
252 // starting at the address of the character pointer passed
253 // -does bounds checking
254
255 Bufr& Bufr::operator=(const char *S)
256 {
257   int tsize;
258   tsize=strlen(S);
259   if( (unsigned)tsize+1 > bufsize)
260     if( resize(tsize+1)!= 0)
261       {
262         //Log( ER_NOMEM, "Unable to resize Bufr" );
263         return *this;
264       }
265    strcpy( ptr, S);
266    return *this;
267 }
268
269 ////////////////////////////////////////////////////////////////////////
270 // Internal + operator
271 //  Created to add Bufr to Bufr
272
273 Bufr Bufr::operator+(const Bufr& B) const
274 {
275   Bufr Tmp( *this );
276
277   unsigned l=size();
278   unsigned l2=B.size();
279
280   if( Tmp.resize( l+l2 ) != 0)
281     {
282       //Log( ER_NOMEM, "Unable to resize Bufr" );
283       return Tmp;
284     }
285   memcpy( (void*) ((char*)Tmp.ptr + l), B.ptr, l2 );
286   return Tmp;
287 }
288
289 ////////////////////////////////////////////////////////////////////////
290 //  Operator to add a string and a buffer
291
292 Bufr Bufr::operator+( const char* S ) const
293 {
294   Bufr Tmp( *this );
295
296   int l=length();
297   int l2=strlen(S);
```



```
298
299   // Extend the buffer if necessary
300   if( (unsigned)l+l2+1 > bufsize )
301     if( Tmp.resize( l+l2+1 ) != 0 )
302       {
303         //Log( ER_NOMEM, "Unable to resize Bufr" );
304         return Tmp;
305       }
306   strcpy( Tmp.ptr, S );
307   return Tmp;
308 }
309
310 ////////////////////////////////////////////////////////////////////////
311
312 Bufr& Bufr::operator+=(const Bufr& B)
313 {
314   int l=size();
315   int l2=B.size();
316   if( resize( l+l2 ) != 0)
317     {
318       //Log( ER_NOMEM, "Unable to resize Bufr" );
319       return *this;
320     }
321   memcpy( ptr+l,B.ptr, l2 );
322   return *this;
323 }
324
325 ////////////////////////////////////////////////////////////////////////
326
327 Bufr& Bufr::operator+=(const char* pstr)
328 {
329   int l=length();
330   int l2=strlen(pstr);
331
332   if( (unsigned)l+l2+1 > bufsize )
333     if( resize( l+l2+1 ) != 0 )
334       {
335         //Log( ER_NOMEM, "Unable to resize Bufr" );
336         return *this;
337       }
338   strcat( ptr,pstr);
339   return *this;
340 }
341
342 ////////////////////////////////////////////////////////////////////////
343
344 ostream& Bufr::printOn( ostream& strm ) const
345 {
346     for(unsigned int i=0; i<=size(); i++ )
347       strm << ptr[i];
348     return strm;
349 };
350
351
352 /*
353   String handling buffer definition
354
355 */
356 ////////////////////////////////////////////////////////////////////////
357
```

```
358 char* strcpy( Sbuf &S, const char *str)
359 {
360   S=str;
361   return(S);
362 }
363
364 void Sbuf::ChangeChr( char from, char to ){
365   char* WordStart;
366   while ( NULL != ( WordStart=strchr( ptr, from )) )
367     *WordStart= to;
368 }
369
370
371 ////////////////////////////////////////////////////////////////////
372 /*
373 char* strcat( Sbuf &S, const char *str)
374 {
375   S=S+str;
376   return(S.Ptr());
377 }
378 */
379
380 ////////////////////////////////////////////////////////////////////
381 // set Sbuf to a another Sbuf
382
383 Sbuf& Sbuf::operator=(Sbuf &S)
384 {
385   if(S.ptr==NULL)
386     {
387       delete[] ptr;
388       ptr=NULL;
389       bufsize=0;
390       return *this;
391     }
392
393   if( bufsize < S.bufsize )
394     if( resize( S.bufsize) != 0 )
395       {
396         //Log( ER_NOMEM, "cannot resize Sbuf" );
397         return *this;
398       }
399   ::strcpy( ptr, S.ptr);
400   return *this;
401 }
402
403 ////////////////////////////////////////////////////////////////////
404 // set Sbuf to a character buffer
405
406 Sbuf& Sbuf::operator=(const char *S)
407 {
408   if( S==NULL)
409     {
410       delete[] ptr;
411       ptr=NULL;
412       bufsize=0;
413       return *this;
414     }
415
416   unsigned int len=strlen(S);
417   if( len+1 > bufsize)
```



```
418     if( resize( len+1) != 0)
419       {
420         //Log( ER_NOMEM, "cannot resize Sbuf" );
421         return *this;
422       }
423   ::strcpy( ptr, S);
424   // Clear the rest of the buffer
425   memset( ptr + len, 0, bufsize - len );
426   return *this;
427 }
428
429
430 ////////////////////////////////////////////////////////////////////
431 // set Sbuf to a character
432
433 Sbuf& Sbuf::operator=(const char S)
434 {
435   if( bufsize < 2 )
436     if(resize(2) != 0)
437       {
438         //Log( ER_NOMEM, "cannot resize Sbuf" );
439         return *this;
440       }
441   ptr[0]=S;
442   ptr[1]=0;
443   return *this;
444 }
445
446 Sbuf& Sbuf::operator+=( const char c ) {
447   // Convert the character to a string
448   Sbuf Tmp;
449   Tmp[ 0 ] = c;
450   // Use the string extension operator
451   return operator+=( Tmp );
452 }
453
454
455 ////////////////////////////////////////////////////////////////////
456 // Sbuf = Sbuf +  a character buffer
457
458 Sbuf& Sbuf::operator+=(const char* pstr)
459 {
460   if(pstr==NULL)
461     return *this;
462
463   int l=length();
464   int l2=strlen(pstr);
465
466   if( (unsigned)l+l2+1 > size() )
467     {
468       if( resize( l+l2+1 ) != 0 )
469         {
470           //Log( ER_NOMEM, "cannot resize Sbuf" );
471           return *this;
472         }
473     }
474
475   ::strcat( ptr, pstr );
476   return *this;
477 }
```

```
478
479 ////////////////////////////////////////////////////////////////////////////
480 // add Sbuf + Sbuf
481
482 Sbuf Sbuf::operator+(Sbuf& B) const
483 {
484   Sbuf Tmp( *this );
485
486   if(B.ptr==NULL)
487     return Tmp;
488
489   int lelength();
490   int l2=B.length();
491
492   if( (unsigned) l+l2+1 > size() )
493     if( Tmp.resize( l+l2+1 ) != 0)
494       {
495         //Log( ER_NOMEM, "cannot resize Sbuf" );
496         return Tmp;
497       }
498   ::strcat( Tmp.ptr, B.ptr );
499   return Tmp;
500 }
501
502 ////////////////////////////////////////////////////////////////////////////
503 // Operator to add a string and a buffer
504
505 Sbuf Sbuf::operator+( const char* S ) const
506 {
507   Sbuf Tmp( *this );
508
509   if(S==NULL)
510     return Tmp;
511
512   int l=length();
513   int l2=strlen(S);
514
515   if (unsigned)l+l2+1 > size() )
516     if( Tmp.resize( l+l2+1 ) != 0)
517       {
518         //Log( ER_NOMEM, "cannot resize Sbuf" );
519         return Tmp;
520       }
521
522   ::strcat( Tmp.ptr, S );
523   return Tmp;
524 }
525
526 ////////////////////////////////////////////////////////////////////////////
527 // add Sbuf + char
528
529 Sbuf Sbuf::operator+(const char B) const
530 {
531   Sbuf Tmp( *this );
532   unsigned int l=length();
533
534   if( l+2 > size() )
535     if((Tmp.resize( l+2 ) != 0)
536       {
537         //Log( ER_NOMEM, "cannot resize Sbuf" );
538         return Tmp;
539       }
540   Tmp.ptr[l]=B;
541   Tmp.ptr[l+1]=0;
542   return Tmp;
543 }
544
545
546 ////////////////////////////////////////////////////////////////////////////
547
548 ostream& Sbuf::print(ostream& strm ) const
549 {
550   int len;
551   if( length()==0 )
552     len=0;
553   else
554     len=length()-1;
555
556   if(!ptr)
557     strm << "<NULL>";
558   else
559     {
560       for(int i=0; i<len; i++ )
561         strm << ptr[i];
562     }
563   return strm;
564 }
565
566 //
567 // Return a specific token from the string buffer
568 // GetToken( "Hello there dolly", 2, GetBuf, ' ' );
569 // Puts there into the getbuf area
570 // Null is a disallowed delimiter
571
572 char* Sbuf::GetToken( int Token, char* TokenBuf, char Delimiter )
573 {
574   char* BeginPtr = ptr;
575
576   int CurToken = 1;
577   if ( *BeginPtr == Delimiter )
578     BeginPtr++;  // Move past the delimiter
579
580   while ( CurToken < Token && *BeginPtr ) {
581
582     // Advance to the next delimiter character
583     while ( *BeginPtr && *BeginPtr != Delimiter )
584       BeginPtr++;
585
586     // Move past the delimiter
587     while ( *BeginPtr && *BeginPtr == Delimiter ) {
588       BeginPtr++;
589     }
590
591     if ( *BeginPtr == NULL ) {
592       // the string was exhausted
593       return NULL;
594     }
595     CurToken++;
596   }
597   // Set the end pointer beyond
```

000041

```
598  char* EndPtr = BeginPtr;
599
600  // Advance the end pointer to the next delimiter
601  // or to the end of string
602  while ( *EndPtr && *EndPtr != Delimiter ) {
603    EndPtr++;
604  }
605
606  int TransferSize = (int)(EndPtr - BeginPtr);
607  if ( TransferSize == 0 ) return NULL;
608
609  memcpy( TokenBuf, BeginPtr, TransferSize );
610  // Put a null byte at the end of the transferred string
611  *(TokenBuf + TransferSize ) = 0;
612  return TokenBuf;
613 }
614
```

C:\TAPVI\BUF.CPP

Mon Sep 30 21:33:18 1996

```
 1
 2 #ifndef _BUF_HPP_
 3 #define _BUF_HPP_
 4
 5 #include <iostream.h>
 6 #include <string.h>
 7
 8 #define ERR_BUF_ALLOC ( 1 )
 9 #define ERR_BUF_BOUNDS ( 2 )
10
11 #ifndef DEFAULT_BUFSIZE
12 #define DEFAULT_BUFSIZE ( 256 )
13 #endif
14
15
16
17 class  Buf
18 {
19   protected:
20
21     char *ptr;
22     unsigned bufsize;
23     virtual int Throw( int ErrCode, char* AuxText );
24
25   public:
26
27     friend void * memcpy ( Buf &, const void *, size_t );
28     friend void * memset ( Buf &, int, size_t );
29
30 // The default constructor does not allocate memory
31
32     Buf(void);          // default coonstructor
33     Buf( unsigned i );  // Allocate a buffer of length i
34     Buf( const char*);  // Allocate a buffer large enough for char string
35     Buf( const Buf& );  //  Copy Constructor
36     ~Buf( void );
37
38     int resize(unsigned);
39     char* Ptr() {return ptr;};
40     unsigned size(void) const;
41     unsigned length(void) const;
42     void clear(void) const;
43
44     char& operator()(unsigned);
45     char& operator[](int);
46
47     friend ostream& operator << ( ostream&, const Buf *);
48     friend ostream& operator << (ostream&, const Buf &);
49     virtual ostream& printOn( ostream& OS ) const {
50       OS << ptr;
51       return OS;
52     }
53
54 };
55 class  Bufr : public Buf
56 {
57   public:
58
59     Bufr( void ) : Buf( DEFAULT_BUFSIZE ) {};
60     Bufr( unsigned x) : Buf(x) {};
61     Bufr( const char *C ) : Buf(C) {};
62     ~Bufr(void) {};
63
64
65 //  Buffer addition operator
66     Bufr  operator+ ( const Bufr& ) const;
67     Bufr  operator+ ( const char* ) const;
68
69 // Assignment operators
70     Bufr& operator=(const char *);
71     Bufr& operator=(const Bufr &);
72
73     Bufr& operator+=(const Bufr& );
74     Bufr& operator+=( const char* );
75
76
77 // Implicit  conversion
78     operator void*()  { return( (void*) ptr ); };
79     operator char*()  { return( (char*) ptr ); };
80     operator unsigned char*() { return( (unsigned char*) ptr ); };
81
82     virtual ostream& printOn( ostream& ) const;
83
84 };
85
86 /*
87   Sbuf is a string handling class
88 */
89
90 class  Sbuf : public Buf
91 {
92   public:
93
94     Sbuf(void) : Buf( DEFAULT_BUFSIZE ) {};
95     Sbuf(unsigned x) : Buf(x+1) {}
96     Sbuf(const char* C) : Buf( DEFAULT_BUFSIZE) {
97       this->operator=( C );
98     };
99     //~Sbuf(void) {};
100
101    Sbuf& operator=(const char *);
102    Sbuf& operator=(const char);
103    Sbuf& operator=(Sbuf &);
104    int operator==( const char* Targ ){
105      return (!strcmp( ptr, Targ )) ;
106    }
107    Sbuf& operator+=(const char );
108    Sbuf& operator+=(const char*);
109    Sbuf  operator+( Sbuf& B) const;
110    Sbuf  operator+( const char*) const;
111    Sbuf  operator+(const char ) const;
112
113    operator char*()        { return( (char *)ptr ); }; // Implicit  conversion
114    virtual ostream& printOn( ostream& ) const;
115
116    void AddTrailer( char c )       {
117      char* LastChr = ptr + strlen( ptr ) - 1;
118      if ( *LastChr != c ) {
119        *(++LastChr) = c;
120        *(++LastChr) = 0;
```

```
 1 ////////////////////////////////////////////////////////////////////
 2 // coptrb.cpp: Cache base class methods.
 3 // Contains code not dependant of data type.
 4 // Copyright(c) 1993 Azarona Software. All rights reserved.
 5 ////////////////////////////////////////////////////////////////////
 6 #include <iostream.h>
 7 #include <fstream.h>
 8 #include "cacheb.h"
 9 #include "bucketb.h"
10
11 void Cacheb::Setup(Bucketb *b, int n, unsigned bkt_size)
12 // Using the n buckets pointed to by b, which are allocated
13 // and constructed, this routine finishes the construction
14 // by building a doubly-linked circular list out of them,
15 // and then configuring for an empty cache.
16 {
17   Bucketb *p = b;
18   for (int i = 0; i<n; i++) {
19       p->MakeNull();
20       p->prev = (Bucketb *)((char *)p - bkt_size);
21       p->next = (Bucketb *)((char *)p + bkt_size);
22       p = p->next;
23   }
24   // Make the list circular
25   Bucketb *tail = (Bucketb *)((char *)b + (n-1)*bkt_size);
26   tail->next = b;
27   b->prev = tail;
28   // Set up other cache variables
29   nbuckets = n; head = b;
30   hits = 0; misses = 0; fast_binds = 0;
31 }
32
33 Cacheb::Cacheb(Bucketb *b, int n, unsigned bkt_size)
34 // Using the n buckets pointed to by b, which are allocated
35 // and constructed, this routine finishes the construction
36 // by building a doubly-linked circular list out of them,
37 // and then configuring for an empty cache.
38 : fptr(0)
39 {
40   Setup(b, n, bkt_size);
41 }
42
43 void Cacheb::Connect(fmgrPtr &f)
44 // Connects cache to file f, assumed already opened.
45 {
46   Clear(); // Flush any pending data
47   fptr = f;
48 }
49
50 void Cacheb::Disconnect()
51 // Disconnects cache from the file it was connected to.
52 {
53 //C700: if (fptr) Clear();  // Flush any pending data
54   if (fptr!=0) Clear(); // Flush any pending data
55   fptr = 0; // Connect to null object
56 }
57
58 void Cacheb::MoveToFront(Bucketb *b)
59 // Logically move bucket to front of list, so that it is
60 // treated as the most recently reserved bucket.
61 // Shortcuts are used when possible.
62 {
63   if (b == head->prev) { // b is tail
64       head = head->prev;
65   }
66   else if (b != head) {
67     // Unhook bucket from where it is
68     b->prev->next = b->next;
69     b->next->prev = b->prev;
70     // Put it before head
71     b->next = head;
72     b->prev = head->prev;
73     b->prev->next = b;
74     head->prev = b;
75     head = b;
76   }
77 }
78
79 void Cacheb::Flush(int empty_bkts)
80 // Flushes all buckets in the cache. Makes them empty if
81 // empty_bkts is true. Checks for dangling coptrs.
82 {
83   // Flush, starting at the front
84   Bucketb *b = head;
85   do {
86     if (empty_bkts && b->IsLocked()) excp->VT_THROW(DANGLINGPTR);
87     if (fptr->ReadyForWriting()) b->Flush(*fptr);
88     if (empty_bkts) b->MakeNull();
89     b = b->next;
90   } while(b != head);
91 }
92
93
94 void Cacheb::Clear()
95 // Flush all buckets in the cache, and then null them out.
96 // An exception is thrown if any dangling coptrs are found.
97 {
98 //C700: if (fptr) Flush(1);
99   if (fptr!=0) Flush(1);
100  hits = 0; misses = 0; fast_binds = 0;
101 }
102
103 Bucketb *Cacheb::AcquireBkt()
104 // Finds least recently reserved bucket that isn't locked,
105 // flushes the bucket, and moves it to the front. Passes
106 // back pointer to bucket or returns 0 if no bucket available.
107 {
108   Bucketb *b = head->prev; // Least-recently reserved bucket
109   while(1) {
110     if (!b->IsLocked()) break;
111     b = b->prev;
112     if (b == head->prev) {
113       excp->VT_THROW(CACHEFULL); // Most likely will exit program
114       return 0;                  // Make compiler happy
115     }
116   }
117   // Flush any data that might be in the bucket and move it to
118   // the front so it becomes the most-recently reserved bucket.
119   b->Flush(*fptr);
120   MoveToFront(b);
```

```
121   return b;
122 }
123
124 Bucketb *Cacheb::FindBkt(long addr)
125 // Searches for a bucket in the cache connected to file address
126 // addr. Returns pointer or 0 if no such bucket found.
127 {
128   // Start search from front, (most recently reserved bucket)
129   Bucketb *b = head;
130   do {
131     if (b->addr == addr) return b;
132     b = b->next;
133   } while(b != head);
134   return 0;
135 }
136
137 Bucketb *Cacheb::ReserveBkt(long addr, int ensure_loaded)
138 // Reserves a bucket for file address addr. If
139 // ensure_loaded == 1, then if the data at addr is not
140 // already in bucket, it is loaded in. Moves bucket to the
141 // front. Passes back pointer to bucket, which may be 0
142 // if there is an error.
143 {
144   Bucketb *b;
145
146   if (addr == 0) return 0; // No sense reserving bucket
147
148   b = FindBkt(addr);
149   if (b == 0) { // Bkt data not loaded, so acquire a bucket
150     misses++;
151     b = AcquireBkt();
152     if (b) {
153       b->addr = addr;
154       if (ensure_loaded && addr) {
155         b->Fetch(*fptr);
156       }
157       // If failure fetching data, return null bucket to
158       // signal failure, else lock the bucket.
159       // Note: Any exception will probably cause us to
160       // exit the program.
161       if (!fptr->Ok()) b = 0; else b->Lock();
162     }
163   }
164   else {
165     hits++;
166     MoveToFront(b);
167     b->Lock();
168   }
169
170   return b;
171 }
172 /*
173 #ifdef OLD
174
175 void Report(Cacheb &c, int full_report)
176 // Friend function that reports on the status of the cache.
177 {
178   Bucketb *b = c.head;
179   int cnt = 0, i = 0;
180   do {
181     if (b->IsLocked()) cnt++;
182     if (full_report) {
183       cout << "Bucket[" << setfill('0') << setw(2) << i << "]: ";
184       cout << "Addr = " << b->addr << ' ';
185       cout << "Refs = " << b->refcnt << ' ';
186       cout << "Dirty = " << int(b->dirty) << '\n';
187     }
188     i++; b = b->next;
189   } while(b != c.head);
190
191   float ratio = 0.0;
192   if (c.hits+c.fast_binds)
193     ratio = ((float)(c.hits+c.fast_binds) /
194       (float)(c.hits+c.fast_binds+c.misses)) * 100.0;
195   cout << "Fast binds/hits/misses: "
196       << c.fast_binds << " / " << c.hits << " / " << c.misses << " ("
197       << setprecision(2) << ratio << "%)\n";
198   cout << "Reservations:          "
199       << (c.hits + c.fast_binds + c.misses) << '\n';
200   cout << "Buckets in use:        "
201       << cnt << " / " << c.nbuckets << "\n";
202 }
203
204 #else
205 */
206
207
208 void Report(Cacheb &c, int full_report)
209 // Friend function that reports on the status of the cache.
210 {
211   Bucketb *b = c.head;
212   int cnt = 0, i = 0;
213
214   if ( full_report ) {
215     cout << "Bucket\tAddr\tRefs\tDirty\t" << endl;
216   }
217
218   do {
219     if (b->IsLocked()) cnt++;
220     if (full_report) {
221       cout << i          << "\t"
222       << b->addr   << "\t"
223       << b->refcnt << "\t"
224       << (int)b->dirty  << "\t"
225       << endl;
226
227       //printf("Bucket[%02d]: Addr = %ld Refs = %d Dirty = %d\n",
228       //       i, b->addr, b->refcnt, b->dirty);
229     }
230     i++; b = b->next;
231   } while(b != c.head);
232
233   long reservations = c.hits + c.fast_binds + c.misses;
234
235   float ratio = 0.0;
236   if (c.hits+c.fast_binds)
237     ratio = (float)((c.hits+c.fast_binds) / (reservations) * 100.0);
238   cout << "Fast binds/hits/misses: "
239       << c.fast_binds << " / " << c.hits << " / " << c.misses << " ("
240       << ratio << "%)" << endl;
```

```
        //<< setprecision(2) << ratio << "%)\n";
    cout << "Reservations:              "
         << (c.hits + c.fast_binds + c.misses) << endl;
    cout << "Buckets in use:            "
         << cnt << " / " << c.nbuckets << endl;

    //printf("Fast binds/hits/misses: %ld / %ld / %ld (%6.2f%)\n",
    //    c.fast_binds, c.hits, c.misses, ratio);
    //printf("Reservations:             %ld\n", reservations);
    //printf("Buckets in use:           %d / %d\n", cnt, c.nbuckets);
}

//#endif
```

```
 1 ////////////////////////////////////////////////////////////////////
 2 // cacheb.h: Resource cache base class definitions.
 3 // This is the data independent code.
 4 // Copyright(c) 1993 Azarona Software. All rights reserved.
 5 ////////////////////////////////////////////////////////////////////
 6 #ifndef H_CACHEB
 7 #define H_CACHEB
 8 #include "fmgr.h"
 9 #include "exchdlr.h"
10
11 class Bucketb; class Copirb; // forward decl's
12
13 class Cacheb {
14 protected:
15   long hits, misses; // For performance statistics
16   long fast_binds;
17   int nbuckets;       // Size of cache
18   FmgrPtr fptr;       // File cache is connected to
19   Bucketb *head;      // Most recently reserved bucket
20   Bucketb *AcquireBkt();
21   Bucketb *FindBkt(long addr);
22   void MoveToFront(Bucketb *b);
23   void Setup(Bucketb *b, int n, unsigned bkt_size);
24   Cacheb(Bucketb *b, int n, unsigned bkt_size);
25   virtual ~Cacheb();
26 public:
27   friend class Copirb;
28   void Connect(FmgrPtr &fp);
29   void Disconnect();
30   void Flush(int empty_bkts = 0);
31   void Clear();
32   virtual Bucketb *ReserveBkt(long addr, int ensure_loaded=1);
33   void FastBind();
34   //C700 ERROR: redefining default parameter 2
35   //C700:  friend void Report(Cacheb &c, int full_report=0);
36   friend void Report(Cacheb &c, int full_report);
37 };
38
39 inline Cacheb::~Cacheb()
40 {
41   // Derived class should do all the work
42 }
43
44 inline void Cacheb::FastBind()
45 {
46   fast_binds++;
47 }
48
49 #endif
```

```
 1 ////////////////////////////////////////////////////////////////////
 2 // coptrb.cpp: Cached object pointer base class methods.
 3 // Does all the things that don't depend on the data stored.
 4 // Copyright(c) 1993 Azarona Software. All rights reserved.
 5 ////////////////////////////////////////////////////////////////////
 6 #include "coptrb.h"
 7 #include "bucketb.h"
 8 #include "cacheb.h"
 9
10 Coptrb::Coptrb(Cacheb &c, long p)
11 // General coptr constructor. Note that a cache
12 // is required. The coptr stays unbound till needed.
13 {
14   cache = &c; addr = p; bound = 0; bkt = 0;
15 }
16
17 Coptrb::Coptrb(const Coptrb &c)
18 // Copy constructor. Note that destination coptr
19 // won't be bound until needed, even if source is bound.
20 {
21   addr = c.addr; bkt = c.bkt; cache = c.cache; bound = 0;
22 }
23
24 Bucketb *Coptrb::Alloc(size_t n, long p)
25 // Allocates n bytes of room (presumably the size of the
26 // data in the bucket) at address p in the cache's
27 // file, and sets up a cache bucket to support it.
28 // If p != 0, it means the room has already been allocated.
29 // Note that ReserveBkt will setup bucket's address,
30 // cache pointer and refcnt, but doesn't load any data.
31 // Returns a pointer to the bucket, or 0 if couldn't
32 // allocate.
33 {
34   Release(); // Let go of any currently bound data
35   if (p == 0) p = cache->fptr->Alloc(n);
36   bkt = cache->ReserveBkt(p, 0);
37   if (bkt) {
38     bkt->SetDirty(); // Data modified by constructor
39     addr = p;
40     bound = 1;
41   }
42   else {
43     addr = 0; // bound will = 0 here as well
44   }
45   return bkt;
46 }
47
48 void Coptrb::Delete(unsigned n)
49 // Deallocates the file data pointed to by this coptr, supposedly
50 // of size n. Deallocation only takes place if this coptr is
51 // pointing to a bucket bound by no one else. Coptr is unbound
52 // afterwards and both the bucket and Coptr are made null.
53 {
54   if (bound && bkt->IsLocked() == 1 ||
55       !bound && (!bkt || !bkt->IsLocked())) {
56     cache->fptr->Free(n, addr);
57     Release();
58     bkt->MakeNull();
59     addr = 0;
60   }
61   else excp->VT_THROW(DANGLINGPTR);
62 }
63
64 void Coptrb::Copy(const Coptrb &c)
65 // Copies one coptr into another. Note that the destination
66 // coptr isn't bound, even if the source coptr is.
67 {
68   Release();
69   addr = c.addr;
70   bkt = c.bkt;
71   bound = 0;
72   cache = c.cache;
73 }
74
75 void Coptrb::Bind()
76 // Binds the coptr to a bucket containing data stored at
77 // file location addr. Note that we may already be pointing
78 // to the correct bucket, so check for that.
79 // ASSUMES not already bound. For internal use only.
80 {
81   if (bkt && bkt->addr == addr) { // Fast binding
82     if (addr) {
83       bkt->Lock();
84       cache->FastBind();
85       bound = 1;
86     }
87     else excp->VT_THROW(NULLPTR);
88   }
89   else {
90     bkt = cache->ReserveBkt(addr, 1);
91     if (bkt) bound = 1; else excp->VT_THROW(NULLPTR);
92   }
93 }
94
95 void Coptrb::Release()
96 // Signal to the bucket that we're through with it.
97 {
98   if (bound) {
99     bkt->Unlock();
100    bound = 0;
101  }
102 }
103
104 void Coptrb::NotifyUseOf()
105 // Moves the referenced bucket to the front of the
106 // cache queue. For cache performance tuning.
107 {
108   if (bound) cache->MoveToFront(bkt);
109 }
```

```
1 ////////////////////////////////////////////////////////////////////////////
2 // coptrb.hi Cached object pointer base class definition.  95TTF
3 // Does all the things that don't depend on the data stored.
4 // Copyright(c) 1993 Asarona Software. All rights reserved.
5 ////////////////////////////////////////////////////////////////////////////
6 #ifndef H_COPTRB
7 #define H_COPTRB
8
9 class Bucketb; class Cacheb; // Forward decl's
10
11 class Coptrb {
12 protected:
13   long addr;      // File address of bucket data
14   Bucketb *bkt;   // Pointer to bucket in the cache
15   Cacheb *cache;  // Pointer to the cache being used
16   char bound;     // True if pointer is bound to a bucket
17   void Copy(const Coptrb &c);
18   Bucketb *Alloc(unsigned n, long p=0);
19   void Delete(unsigned n);
20   void Bind();
21   Coptrb(Cacheb &c, long p);
22   Coptrb(const Coptrb &c);
23   void operator=(const Coptrb &c) { }; // Disallowed
24 public:
25   ~Coptrb();
26   void Grab();
27   void Release();
28   void NotifyUsed();
29   operator long() const;
30 };
31
32 inline Coptrb::~Coptrb()
33 // We know we can release the bucket when we're through
34 // with the coptr.
35 {
36   Release();
37 }
38
39 inline Coptrb::operator long() const
40 // We let a coptr look like a file address, since in a way
41 // it's just a smart version of a file address.
42 {
43   return addr;
44 }
45
46 inline void Coptrb::Grab()
47 // Like Bind(), but makes no assumptions about the coptr
48 // being unbound coming in. For public use.
49 {
50   if (!bound) Bind();
51 }
52
53 #endif
```

```
1 ////////////////////////////////////////////////////////////////////////////
2 // excdir.cpp "Exception handler" class methods.
3 // *** THIS IS A SIOIO VERSION, UNLIKE IOSTREAM VERSION IN BOOK ***
4 // Copyright(c) 1993 Azarona Software. All rights reserved.
5 ////////////////////////////////////////////////////////////////////////////
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <stdio.h>
9 #include "excHdr.h"
10 ExceptionHandler DefExceptionHandler;
11 ExceptionHandler *exep = &DefExceptionHandler;
12
13
14 char *exception_msg[ 7 ] = {
15     "No exception intended, none taken",          // NONE
16     "Dangling 'smart pointer'",                   // DANGLINGPTR
17     "Accessing a null 'smart pointer'",           // NULLPTR
18     "Cache full",                                 // CACHEFULL
19     "Stack full",                                 // STKFULL
20     "Stack empty",                                // STKEMPTY
21     "Assertion failed"                            // ASSERTERR
22 };
23
24 void ExceptionHandler::PrtException()
25 {
26   printf("%s", exception_msg(exception));
27 }
28
29 void ExceptionHandler::PrtExceptionCode ec)
30 {
31   printf("%s", exception_msg(ec));
32 }
33
34 void ExceptionHandler::Report(char *src, int lno,
35     char *msg, char *dev)
36 // Prints out a message corresponding to the exception. If msg
37 // isn't null, it is printed on the next line. If dev (presumably
38 // a device name) isn't null, it is printed on the line below msg.
39 // NOTE: Some exception codes don't cause anything to be printed.
40 {
41   if (exception != NONE) {
42     if (src) {
43       printf("\nEXCEPTION ON line %d of '%s':\n", lno, src);
44       printf("%s\n", exception_msg(exception));
45     }
46     else {
47       printf("\nEXCEPTION: %s\n", exception_msg(exception));
48     }
49     if (msg) printf("%s\n", msg);
50     if (dev) printf("Error occurred on: %s\n", dev);
51   }
52   fflush(stdout);
53 }
54
55 void ExceptionHandler::Throw(ExceptionCode ec, char *src,
56     int lno, char *msg, char *dev)
57 // Prints out a message corresponding to the exception code.
58 // then the exception is handled.
59 {
60   exception = ec;
```

000051

```
1 ////////////////////////////////////////////////////////////////////////
2 // exchdlr.h: "Exception handler" class definition.
3 // Notes: We are in no way trying to emulate C++ exception
4 // handling. The class defined allows us to conveniently
5 // abort the program when an error occurs.
6 // Copyright(c) 1993 Azarona Software. All rights reserved.
7 ////////////////////////////////////////////////////////////////////////
8 #ifndef H_EXCHDLR
9 #define H_EXCHDLR
10
11 #define VT_THROW(c) Throw(c, __FILE__, __LINE__)
12
13 enum ExceptionCode {
14   NONE,
15   DANGLINGPTR,
16   NULLPTR,
17   CACHEFULL,
18   STKFULL,
19   STKEMPTY,
20   ASSERTERR,
21   NUM_CODES   // Neat trick
22 };
23
24 class ExceptionHandler {
25 protected:
26   virtual void Report(char *src, int lno, char *msg, char *dev);
27   virtual void TakeAction();
28 public:
29   ExceptionCode exception;
30   ExceptionHandler();
31   void Clear();
32   void PrtException();
33   void PrtException(ExceptionCode ec);
34   virtual void Throw(ExceptionCode ec,
35     char *src=0, int lno=0, char *msg=0, char *dev=0);
36 };
37
38 inline ExceptionHandler::ExceptionHandler()
39 {
40   exception = NONE;
41 };
42
43 inline void ExceptionHandler::Clear()
44 {
45   exception = NONE;
46 };
47
48 extern ExceptionHandler DefExceptionHandler;
49 extern ExceptionHandler *exep;
50
51 #endif
52
```

_equibm C:\TMP\VT\EXCHDLR.H Mon Feb 19 09:19:54 1996

NOV 14 '97 16:44 FR BROOKS & MCKENZIE 212 759 9133 TO 15561045011391160 P.86

```
 1 #include <exception.h>
 2 #include <stdlib.h>
 3
 4 // the one and only definition of errno
 5 // only needed if we are doing a multi-threaded build
 6
 7 void Exception::Report() {
 8   /*
 9     if ( FunctionName )
10       cerr << "Exception thrown by:  " << FunctionName << endl;
11
12     if ( ExceptionCode ) {
13       cerr << "Exception Code:  " << ExceptionCode << endl;
14     }
15
16     if ( ErrCode )
17       cerr << "Error Code:           " << ErrCode << endl;
18   */
19   cerr << GetReportString();
20
21   if ( ExceptionClassCode )
22     cerr << "Exception Class Code:  " << ExceptionClassCode << endl;
23
24   if ( ErrCode )
25     cerr << "Error Code:  " << endl;
26
27 #ifdef __BORLANDC__
28     cerr << "Exception Name:       " << __throwExceptionName << endl;
29     cerr << "Module Name:          " << __throwFileName      << endl;
30     cerr << "Line Number:          " << __throwLineNumber    << endl;
31 #endif
32
33 // Multitasking libraries do not support errno as an int
34 #if !defined( NO_ERRNO )
35   if ( errno )
36     ErrnoReport();
37 #endif
38 }
39
40 void ErrnoReport() {
41 #if !defined( NO_ERRNO )
42   cerr << "Error Number:  " << errno << "  ";
43
44 # ifdef __BORLANDC__
45   switch ( errno ) {
46     case 0: break;
47     case EINVFNC:  cerr << "Invalid function number"  << endl;
48                    break;
49     case ENOENT:   cerr << "No such file or directory"  << endl;  break;
50     case ENOPATH:  cerr << "Path not found"  << endl;  break;
51     case EMFILE:   cerr << "Too many open files"  << endl;  break;
52     case EACCES:   cerr << "Permission denied"  << endl;  break;
53     case EBADF:    cerr << "Bad file number"  << endl;  break;
54     case ECONTR:   cerr << "Memory blocks destroyed "  << endl;  break;
55     case ENOMEM:   cerr << "Not enough core"  << endl;  break;
56     case EINVMEM:  cerr << "Invalid memory block address "  << endl;
57                    break;
58     case EINVENV:  cerr << "Invalid environment"  << endl;  break;
59     case EINVFMT:  cerr << "Invalid format"  << endl;  break;
60     case EINVACC:  cerr << "Invalid access code"  << endl;  break;
61     case EINVDAT:  cerr << "Invalid data"  << endl;  break;
62     case EINVAL:   cerr << "Unknown error"  << endl;  break;
63     case ENODEV:   cerr << "No such device"  << endl;  break;
64     case ECURDIR:  cerr << "Attempt to remove CurDir"  << endl;  break;
65     case ENOTSAM:  cerr << "Not same device"  << endl;  break;
66     case ENMFILE:  cerr << "No more files"  << endl;  break;
67     case EINVAL:   cerr << "Invalid argument"  << endl;  break;
68     case E2BIG:    cerr << "Arg list too long"  << endl;  break;
69     case ENOEXEC:  cerr << "Exec format error"  << endl;  break;
70     case EXDEV:    cerr << "Cross-device link"  << endl;  break;
71     case ENFILE:   cerr << "Too many open files"  << endl;  break;
72     case ECHILD:   cerr << "No child process"  << endl;  break;
73     case ENOTTY:   cerr << "UNIX - not MSDOS"  << endl;  break;
74     case ETXTBSY:  cerr << "UNIX - not MSDOS"  << endl;  break;
75     case EFBIG:    cerr << "UNIX - not MSDOS"  << endl;  break;
76     case ENOSPC:   cerr << "No space left on device"  << endl;  break;
77     case ESPIPE:   cerr << "Illegal seek"  << endl;  break;
78     case EROFS:    cerr << "Read-only file system"  << endl;  break;
79     case EMLINK:    cerr << "UNIX - not MSDOS"  << endl;  break;
80     case EPIPE:    cerr << "Broken pipe"  << endl;  break;
81     case EDOM:     cerr << "Math argument"  << endl;  break;
82     case ERANGE:   cerr << "Result too large"  << endl;  break;
83     case EEXIST:   cerr << "File already exists"  << endl;  break;
84     case EDEADLOCK: cerr << "Locking violation"  << endl;  break;
85     case EPERM:     cerr << "Operation not permitted"  << endl;  break;
86     case ESRCH:    cerr << "UNIX - not MSDOS"  << endl;  break;
87     case EINTR:    cerr << "Interrupted function call"  << endl;  break;
88     case EIO:      cerr << "Input/output error"  << endl;  break;
89     case ENXIO:    cerr << "No such device or address"  << endl;  break;
90     case EAGAIN:   cerr << "Resource temporarily unavailable"  << endl;
91                    break;
92     case ENOTBLK:  cerr << "UNIX - not MSDOS"  << endl;  break;
93     case EBUSY:    cerr << "Resource busy"  << endl;  break;
94     default:
95       cerr << "Unrecognize Global Error:  " << errno << endl;
96       break;
97   }
98 # endif
99
100 # if defined( WIN32 )
101   switch ( errno ) {
102     case 0: break;
103     case EPERM:     cerr << "EPERM" << endl; break;
104     case ENOENT:    cerr << "ENOENT" << endl; break;
105     case ESRCH:     cerr << "ESRCH" << endl; break;
106     case EINTR:     cerr << "EINTR" << endl; break;
107     case EIO:       cerr << "EIO" << endl; break;
108     case ENXIO:     cerr << "ENXIO" << endl; break;
109     case E2BIG:     cerr << "E2BIG" << endl; break;
110     case ENOEXEC:   cerr << "ENOEXEC" << endl; break;
111     case EBADF:     cerr << "EBADF" << endl; break;
112     case ECHILD:    cerr << "ECHILD" << endl; break;
113     case EAGAIN:    cerr << "EAGAIN" << endl; break;
114     case ENOMEM:    cerr << "ENOMEM" << endl; break;
115     case EACCES:    cerr << "EACCES" << endl; break;
116     case EFAULT:    cerr << "EFAULT" << endl; break;
117     case EBUSY:     cerr << "EBUSY" << endl; break;
118     case EEXIST:    cerr << "EEXIST" << endl; break;
119     case EXDEV:     cerr << "EXDEV" << endl; break;
120     case ENODEV:    cerr << "ENODEV" << endl; break;
```

```
121     case ENOTDIR:   cerr << "ENOTDIR" << endl; break;
122     case EISDIR:    cerr << "EISDIR" << endl; break;
123     case EINVAL:    cerr << "EINVAL" << endl; break;
124     case ENFILE:    cerr << "ENFILE" << endl; break;
125     case EMFILE:    cerr << "EMFILE" << endl; break;
126     case ENOTTY:    cerr << "ENOTTY" << endl; break;
127     case EFBIG:     cerr << "EFBIG" << endl; break;
128     case ENOSPC:    cerr << "ENOSPC" << endl; break;
129     case ESPIPE:    cerr << "ESPIPE" << endl; break;
130     case EROFS:     cerr << "EROFS" << endl; break;
131     case EMLINK:    cerr << "EMLINK" << endl; break;
132     case EPIPE:     cerr << "EPIPE" << endl; break;
133     case EDOM:      cerr << "EDOM" << endl; break;
134     case ERANGE:    cerr << "ERANGE" << endl; break;
135     case EDEADLK:   cerr << "EDEADLK" << endl; break;
136     case ENAMETOOLONG:   cerr << "ENAMETOOLONG" << endl; break;
137     case ENOLCK:    cerr << "ENOLCK" << endl; break;
138     case ENOSYS:    cerr << "ENOSYS" << endl; break;
139     case ENOTEMPTY: cerr << "ENOTEMPTY" << endl; break;
140     case EILSEQ:    cerr << "EILSEQ" << endl; break;
141     default:
142        cerr << "Unrecognized MSC error number" << endl;
143        break;
144   }
145 # endif WIN32
146 #endif // NO_ERRNO
147 }
148 /*
149    Default crash function with reporting
150    Use this line in main to put this function in place:
151       set_terminate( TerminalError );
152    called for uncaught errors
153 */
154 void TerminalError() {
155    cerr << "Uncaught Terminal Error" << endl;
156
157 # ifdef __BORLANDC__
158    cerr << "Exception Name:  " << __throwExceptionName << endl;
159    cerr << "Module Name:     " << __throwFileName      << endl;
160    cerr << "Line Number:     " << __throwLineNumber    << endl;
161 # endif
162
163 #if !defined( NO_GUARD )
164    if ( errno ) {
165       ErrnoReport();
166    }
167 #endif
168    abort();
169 }
```

```
/*
  Exception handler for DI API

  The objective if this class is to provide basic handling for
  thrown exceptions not caught by lower levels.
 */

#ifndef _EXCEPTION_HPP_
#define _EXCEPTION_HPP_

// export all classes if we are building a dll
#if defined( SSK_DLL )
#include "asksglob.h"
#else
#define EXPORTSPEC
#endif

#if defined( __BORLANDC__ )
#include <except.h>
#endif

#if defined( _DOS) || defined(_WINDOWS) || defined( WIN32 )
#include <eh.h>
#include <stdlib.h>
#endif

#include <iostream.h>
#include <string.h>

// Bring in the system error info
#if !defined( NO_ERRNO )
#include <errno.h>
extern int errno;
#endif

// Translate the error number
void ErrnoReport();

// Standard service functions
void TerminalError();  // Report an uncaught errors and abort

/*
// This class redirects the new handler to throw a memory error
//  when an allocation fails.  When the NewHandler goes out of
//  scope, the new handler is set back to the default
class NewHandler {
  new_handler (*SaveHandler ) ();
 public:
   NewHandler( (void)(*ReplacementHndlr) = NULL ) {
    SaveHandler = set_new_handler ( ReplacementHndlr );
   }
}
*/

// The exception code is an object identifyer which enables
// a general purpose exception catch to determine what
// type of exception was thrown.
#define NULL_EXCEPTION_CODE ( 0 )
class EXPORTSPEC Exception {
 protected:
  char  FunctionName[ 64 ];  // Pointer to message string
  int   ErrCode;             // Pass a numeric key
  int   ExceptionClassCode;  // Object key which threw error
 public:
  Exception() {
    FunctionName[ 0 ] = 0;
    ErrCode = 0;
    ExceptionClassCode = 0;
  }
  Exception( char* Str ) {
    strcpy( FunctionName, Str );
    ErrCode = 0;
    ExceptionClassCode = 0;
  }

  // Resetting the errno to zero may be inappropriate
  // because destruction of a derived object may wipe out
  // info we need to report the error at a higher level
  ~Exception( ) { /* errno = 0; */ }

  void virtual Report();

  // This virtual function assembles the error report
  // into a string to enable the application to display
  // it without having to derive from the exception class
  virtual char* GetReportString() {
    // Make the buffer static to prevent it from
    // being deleted after returning
    static char ReportString[ 64 ];
    strcpy( ReportString, FunctionName );
    return ReportString;
  }

  // Set and retrieve the class identifyer for the
  // class which is generating the exception.
  int  GetExceptionClassCode(){
    return ExceptionClassCode;
  }
  void SetExceptionClassCode( int ClassCode ) {
    ExceptionClassCode = ClassCode;
  }

  // Set and retrieve the errorcode which is specific
  // to the class
  void SetErrCode( int EC ) { ErrCode = EC; }
  int  GetErrCode( ) { return ErrCode; }

  void SetFunctionName( char* Msg ) {
    if ( strlen( Msg ) > 64 ) {
      cerr << "Function Name too large for error buffer" << endl;
      return;
    }
    strcpy( FunctionName, Msg );
  }

  void Clear() {
    memset( FunctionName, 0, 64 );
    ErrCode = 0;
    // Do not reset Exception code
  }
```

```
121
122 );                                                          SCOTT
123
124 /*
125   This class is intended to project file specific errors to
126   higher levels.
127   In certain circumstances, errors may be handled by parental
128   elements such that an application can continue.
129
130   In other cases it may represent a serious error resulting in a termination
131   or an abort
132
133   The error codes are intended to assist in determining the error type
134   however, the errno may also be useful in diagnosing problems
135 */
136 #define FILE_EXCEPTION_CODE ( 1 )
137
138 #define FILE_OPEN_ERROR     ( 1 )
139 #define FILE_CLOSE_ERROR    ( 2 )
140 #define FILE_SEEK_ERROR     ( 3 )
141 #define FILE_READ_ERROR     ( 4 )
142 #define FILE_WRITE_ERROR    ( 5 )
143
144 #define FILE_CREATE_ERROR   ( 6 )
145
146 class EXPORTSPEC FileError : public Exception {
147   char FileName[ 128 ];
148 public:
149   FileError( char* Function, char* Fname= NULL ) : Exception( Function ) {
150     if ( Fname )
151       strcpy( FileName, Fname );
152     else
153       FileName[ 0 ] = 0;
154
155     ExceptionClassCode = FILE_EXCEPTION_CODE;
156   };
157   void SetFileName( char* FN ) { strcpy( FileName,FN ); };
158
159   virtual char* GetReportString() {
160     static char ReportString[ 256 ];
161     memset( ReportString, 0, 256 );
162
163     strcpy( ReportString, "File Error Thrown\n" );
164
165     if ( strlen( FunctionName ) ) {
166       strcpy( ReportString, FunctionName );
167       strcat( ReportString, "\n" );
168     }
169
170     if ( strlen( FileName ) ){
171       strcat( ReportString, FileName );
172       strcat( ReportString, "\n" );
173     }
174     return ReportString;
175   }
176 };
177
178 /*
179   ProgramError
180   This class is intended as a catch for assert style programming.
```



```
181   These errors are always fatal and call terminate.  They result
182   from an illegal condition and usually if not always indicate a
183   bug.                                                  CLASSES
184 */
185 #define PROGRAM_EXCEPTION_CODE ( 2 )
186
187 class EXPORTSPEC ProgramError : public Exception {
188 protected:
189   char Detail[ 128 ];   // What probably went wrong
190 public:
191   ProgramError( char* Msg, char* Msg1 = NULL ) : Exception( Msg ) {
192     ExceptionClassCode = PROGRAM_EXCEPTION_CODE;
193     if ( Msg1 )
194       strcpy( Detail, Msg1 );
195     else
196       Detail[ 0 ] = 0;
197   }
198
199   virtual char* GetReportString() {
200     static char ReportString[ 256 ];
201     memset( ReportString, 0, 512 );
202
203     strcpy( ReportString, "Program Error Thrown\n" );
204
205     if ( strlen( FunctionName ) ) {
206       strcat( ReportString, FunctionName );
207       strcat( ReportString, "\n" );
208     }
209
210     if ( strlen( Detail ) ) {
211       strcat( ReportString, Detail );
212       strcat( ReportString, "\n" );
213     }
214
215     return ReportString;
216   }
217
218 };
219 /*
220   Note:
221     Borland throws xalloc as part of the default new handler.
222     This class is redundant with the default strategy which is presumably
223     part of the Ansi C++ standard.
224
225   Warning:
226     Microsoft's implentation of xalloc requires use of the new handler
227     to throw an xalloc error.  New failures will call new handler
228     under microsoft.
229
230 */
231
232
233 /*
234   this is a working list of the exception codes in use
235   when you define a new exception class put an entry here
236   so that the next time you know where to begin
237 */
238 #define NULL_EXCEPTION_CODE ( 0 )
239 #define FILE_EXCEPTION_CODE ( 1 )
240 #define PROGRAM_EXCEPTION_CODE ( 2 )
```

000056

Mon Aug 26 11:20:01 1996

C:\TMP\VT\EXCPTION.H

squibb

```
261 #define FILENOM_EXCEPTION_CODE ( 3 )
262 #define RollbackJournal_EXCEPTION_CODE ( 4 )
263 #define TABLENOM_EXCEPTION_CODE ( 5 )
264 #define RELATION_EXCEPTION_CODE ( 6 )
265 #define VTFILE_EXCEPTION_CODE   ( 7 )
266 #define TREEDATA_EXCEPTION_CODE      ( 8 )
267 #define TERAS_EXCEPTION_CODE ( 10 )
268 #define BRIDGE_EXCEPTION_CODE ( 11 )
269
270 #define FILEITERATOR_EXCEPTION_CODE ( 12 )
271
272 // Service Table drivers for SSK
273 #define SSK_IBLDRIVER_EXCEPTION_CODE (13)
274 #endif
```

96TT1 8600E09

NOV 14 '97 16:45 FR BAKER & MCKENZIE 212 759 9133 TO 15561045011#9160 P.91

```
 1 ////////////////////////////////////////////////////////////////////////
 2 // fmgr.cpp: File manager class methods.
 3 // Copyright (c) 1989-1993 Azarena Software. All rights reserved.
 4 ////////////////////////////////////////////////////////////////////////
 5 #include <string.h>
 6 #include <stdlib.h>
 7 #include <errno.h>
 8 #include "fmgr.h"
 9
10 // Allocate all necessary static data
11
12 char fmgr::sg[4] = {'F', 'M', 'G', 'R'};
13
14 ////////////////////////////////////////////////////
15 // fmgr methods
16 ////////////////////////////////////////////////////
17
18 fmgr::fmgr()
19 // Creates a file manager object.
20 {
21   // Override default xlat type. We almost always want to
22   // be a binary file here.
23   HeaderDirty = 0;
24
25 }
26
27 fmgr::~fmgr()
28 {
29   // WARNING: Don't rely on base class destructor using
30   // the correct versions of any virtual functions being
31   // called here directly or indirectly. So do your
32   // bid'ness now.
33
34   PrepareToDestroy();
35 }
36
37 int fmgr::__Create(char *fname)
38 {
39   return Create(fname, 0);
40 }
41
42 int fmgr::Create(char *fname, long static_sz)
43 // Creates and opens a new file named fname,
44 // truncating it if it already exists. The
45 // area at the front of the file of length
46 // static_sz + sizeof(Header) is reserved.
47 // Returns exception code.
48 {
49
50   int rv = VtFile::__Create(fname);
51
52   if (rv == VT_SUCCESS) {
53     h.hs = static_sz + sizeof(Header);
54     InitHdr();
55   }
56
57   return rv;
58 }
59
60 void fmgr::InitHdr()
61 // Sets up the header area in the file. If there's
62 // statically allocated data beyond the header, write
63 // a 0-to the last byte of it, so that you won't get
64 // unexpected end of file errors. Note that the rest
65 // of the static area stays uninitialized.
66 {
67   h.fs = 0;
68   h.fe = h.hs;
69   memcpy(h.sg, sg, 4); // Copy signatures
70   memset( &(h.aux), 0, AUXHEADERS * 4 );
71   //h.aux[0] = 0; h.aux[1] = 0;
72   //h.aux[2] = 0; h.aux[3] = 0;
73   WriteHdr();
74   if (h.hs > sizeof(Header)) {
75     char zero_byte = 0;
76     Store(&zero_byte, 1, h.hs-1);
77   }
78 }
79
80 int fmgr::Open(char *fname, AccessMode mode)
81 // Opens the fname file. File must exist and be an fmgr-type
82 // file or error occurs. First closes the current file if open.
83 // Returns exception code. May throw an exception too.
84 {
85   int rv = VtFile::Open(fname, mode);
86
87   if (rv == VT_SUCCESS) {
88     ReadHdr();
89     if (memcmp(h.sg, sg, 4)) { // Test file type
90       Close(0);
91       VtFileError VFE( "VtFile::Open", "Bad file header" );
92       VFE.SetErrCode( FMGR_BAD_FILE_HDR );
93       throw VFE;
94 //        rv = FTYPERR;
95     }
96   }
97
98   return rv;
99 }
100
101 void fmgr::Flush()
102 // Writes the fmgr header to the file, and then flushes
103 // any internal buffers the file might have.
104 {
105   if (ReadyForWriting()) {
106     if ( HeaderDirty ) {
107       WriteHdr();
108       HeaderDirty = 0;
109     }
110     VtFile::Flush();
111   }
112 }
113
114 void fmgr::ReadHdr()
115 // Reads the file header.
116 {
117   Fetch(&h, sizeof(Header), 0L);
118 }
119
120 void fmgr::WriteHdr()
```

```
121 // Writes the file header.
122 {
123   Store(&h, sizeof(Header), 0L);
124 }
125
126 void Fmgr::FetchFBlkHdr(FBlkHeader &h, long p)
127 // Reads in free block header, and tests check word
128 // to make sure the file is still in sync.
129 {
130   Fetch(&h, sizeof(FBlkHeader), p);
131   if (h.check_word != FREEBLK) {
132     VtFileError VFE( "VtFile::FetchFBlkHdr", "Bad free block header" );
133     VFE.SetErrCode( FMGR_BAD_FREEBLK_HDR );
134     throw VFE;
135   }
136 }
137
138 long Fmgr::Reclaim(unsigned nbytes)
139 // Finds the first block on the free space list that is big
140 // enough for nbytes of data. Puts back on the free list
141 // all those bytes that aren't needed. Note that the amount
142 // put back must be at least the size of the free block
143 // header, otherwise, the entire block is considered not
144 // an appropriate size and rejected. Returns address of the
145 // newly reclaimed data, or 0 if there wasn't a free space
146 // block that was appropriate.
147 {
148   FBlkHeader bh, prev_bh, new_bh;
149   long addr, prev_addr, new_addr;
150   unsigned avail_len, unused_len;
151
152   addr = h.fe; prev_addr = 0;
153   HeaderDirty = 1;
154
155   while(addr) { // Until a block of a valid size is found
156     FetchFBlkHdr(bh, addr);
157     if (!Ok()) break;
158     avail_len = bh.len + sizeof(FBlkHeader);
159     if (avail_len >= nbytes + sizeof(FBlkHeader))
160       unused_len = avail_len - nbytes;
161     else unused_len = 0;
162     if (avail_len == nbytes) {
163       // Block is an exact fit, so link prev link to next link
164       if (prev_addr == 0) {
165         // We're at the head of freespace, so we have a new head
166         h.fe = bh.next;
167         HeaderDirty = 1;
168       }
169       else {
170         // In the middle of free space, so link prev to next
171         prev_bh.next = bh.next;
172         StoreFBlkHdr(prev_bh, prev_addr);
173       }
174       break;
175     }
176     else if (unused_len > 0) {
177       // Block too big, and there's room for a free block
178       // header in the unused portion. So splice in this
179       // new block.
180       new_addr = addr + nbytes;
```



```
181       new_bh.check_word = FREEBLK;
182       new_bh.next = bh.next;
183       new_bh.len = unused_len - sizeof(FBlkHeader);
184       StoreFBlkHdr(new_bh, new_addr);
185       if (prev_addr == 0) {
186         // We're at the head of freespace, so we have a new head
187         h.fe = new_addr;
188         HeaderDirty = 1;
189       }
190       else { // In the middle of freespace
191         prev_bh.next = new_addr;
192         StoreFBlkHdr(prev_bh, prev_addr);
193       }
194       break;
195     }
196     else {
197       // Block not big enough, so try next block
198       prev_addr = addr;
199       prev_bh = bh;
200       addr = bh.next;
201     }
202   } // End of looking for big enough block
203
204   return Ok() ? addr : 0;
205 }
206
207
208 long Fmgr::Alloc(unsigned nbytes)
209 // Allocates a block of nbytes of data, either from the free space list,
210 // or from the end of the file. The minimum number of bytes allocated
211 // is equal to sizeof(BlkHeader). Nothing is written to the newly
212 // allocated space.
213 // Returns location of space allocated, or returns a 0 if couldn't
214 // allocate for some reason. An exception may also be thrown.
215 {
216   long p = 0;
217
218   if ( ! ReadyForWriting()) {
219     VtFileError VFE( "VtFiles::Alloc", "File Not opened writable" );
220     VFE.SetErrCode( FMGR_FILE_UNWRITEABLE );
221     throw VFE;
222   }
223
224   // Adjust number of bytes to allocate to minimum size
225
226   if (nbytes < sizeof(FBlkHeader)) nbytes = sizeof(FBlkHeader);
227
228   // Try using a free block
229   p = Reclaim(nbytes);
230
231   if (Ok() && p == 0) { // We may have to extend the file instead
232     p = h.fe;
233     h.fe += nbytes;
234     HeaderDirty = 1;
235     // Write to the last byte to avoid possibe end of file errors
236     char zero = 0;
237     Store(&zero, sizeof(char), h.fe-1);
238   }
239   return p;
240 }
```

```
241
242 void Fmgr::Free(unsigned nbytes, long p)
243 // Frees up the block at location p ASSUMED to be
244 // nbytes in size. Block is placed on the front
245 // of the free space list.
246 // If nbytes is < sizeof(FBlkHeader), it is forced
247 // to that size, since that's the minimum size
248 // allocated.
249 {
250   FBlkHeader fh;
251
252   if (! ReadyForWriting()) {
253     VtFileError VFE( "VtFile::Free", "File Not opened writable" );
254     VFE.SetErrCode( FMGR_FILE_UNWRITEABLE );
255     throw VFE;
256   }
257
258
259     if (nbytes < sizeof(FBlkHeader)) nbytes = sizeof(FBlkHeader);
260     fb.check_word = FREEBLK;
261     fh.len = nbytes - sizeof(FBlkHeader);
262     // Block to become head of free list
263     fb.next = h.fs;
264     StoreFBlkHdr(fh, p);
265     h.fs = p;
266     HeaderDirty = 1;
267 }
268
269
270 ////////////////////////////////////////////////////////////
271 // FmgrPtr Methods
272 ////////////////////////////////////////////////////////////
273
274 void FmgrPtr::Release()
275 {
276   if (objptr) {
277     objptr->refcnt--;
278     if (objptr->refcnt == 0) delete objptr;
279   }
280 }
281
282 void FmgrPtr::Bind(Fmgr *p)
283 {
284   objptr = p;
285   if (objptr) objptr->refcnt++;
286 }
287
288 void FmgrPtr::NewBinding(Fmgr *p)
289 // ASSUMES we'll be owning what p points to.
290 {
291   if (objptr != p) {
292     Release();
293     Bind(p);
294   }
295 }
296
297
298 FmgrPtr::FmgrPtr(Fmgr *p)
299 // ASSUMES we'll be owning what p points to.
300 {
```



```
301   Bind(p);
302 }
303
304 FmgrPtr::FmgrPtr(const FmgrPtr &a)
305 {
306   Bind(a.objptr);
307 }
308
309
310 FmgrPtr::~FmgrPtr()
311 {
312   Release();
313 }
```

```
1 ////////////////////////////////////////////////////////////////////////////
2 // fmgr.h: A file manager class to handle file-based objects.
3 // Copyright(c) 1989-1993 Asgrove Software. All rights reserved.
4 ////////////////////////////////////////////////////////////////////////////
5 #ifndef M_FMGR
6 #define M_FMGR
7
8 #ifndef M_FILE
9 #include <vfile2.h>
10 #endif
11
12
13 // Allocate a fair chunk of space for headers
14 #define AUXHEADERS ( (512 - 16) / 4 )
15 #define MAXINDEXES ( AUXHEADERS / sizeof( FullIdxHdr ) )
16
17 // These are the construction parameters
18 //  for the trees.  They are only needed
19 //  during creation and should be static thereafter
20 //  12 bytes total
21
22 struct FullIdxHdr {
23     long  Offset;
24     short NodeSize;
25     short MaxKeySize;
26     short CacheSize;
27     short MaxHeight;
28     void clear() { memset( this, 0, sizeof( FullIdxHdr ) ); }
29 };
30     struct FBlkHeader { // for free blocks
31         int check_word;     // For file integrity checks. Optional.
32         unsigned len;       // Length of block not including header.
33         long next;          // Pointer to next free block
34     };
35     struct Header { // Goes at the beginning of every fmgr file
36         char sg[4];  // Signature used for every fmgr-based file
37         long fs;     // Address to first block of "heap" free space
38         long fe;     // Address of byte after end of file
39         long hs;     // Address of the start of the "heap"
40         union {
41             long aux[AUXHEADERS]; // Auxiliary offsets into the file
42             FullIdxHdr AuxIdx[ MAXINDEXES ]; // Full blown headers
43         };
44     };
45 class fmgr : public vfile {
46 public:
47
48     enum CheckWord { FREEBLK = 0xFEFE };
49
50     friend class fmgrPtr;
51
52     //struct Header { // Goes at the beginning of every fmgr file
53     //    char sg[4];  // Signature used for every fmgr-based file
54     //    long fs;     // Address to first block of "heap" free space
55     //    long fe;     // Address of byte after end of file
56     //    long hs;     // Address of the start of the "heap"
57     //    union {
58     //        long aux[AUXHEADERS]; // Auxiliary offsets into the file
59     //        FullIdxHdr AuxIdx[ MAXINDEXES ]; // Full blown headers
60     //    };
61 //};
62 // struct FBlkHeader { // for free blocks
63 //     int check_word;     // For file integrity checks. Optional.
64 //     unsigned len;       // Length of block not including header.
65 //     long next;          // Pointer to next free block
66 // };
67
68
69 protected:
70
71     static char sg[4]; // Signature for fmgr style files
72     Header h;          // File header as stored in memory
73
74     int HeaderDirty;   // Is the header dirty
75
76 protected:
77
78     virtual int __Create(char *fname);
79     void InitHdr();
80     void FetchFBlkHdr( FBlkHeader &h, long p);
81     void StoreFBlkHdr( FBlkHeader &h, long p);
82     long AcceleAlloc(unsigned nbytes);
83
84 public:
85
86     fmgr();
87     virtual ~fmgr();
88     int Create(char *fname, long static_size=0); // Must "not" be virtual
89     virtual int Open(char *fname, AccessMode mode = READ_WRITE);
90     virtual void Flush();
91     long Alloc(unsigned nbytes);
92     void Free(unsigned nbytes, long p);
93
94     void ReadHdr();
95     void WriteHdr();
96
97     long Aux(int i) const;
98     long &Aux(int i);
99     FullIdxHdr FullHdr(int i) const;
100    FullIdxHdr &FullHdr(int i);
101
102    //void ClearHdr() { h.clear(); }
103 };
104
103 inline void fmgr::StoreFBlkHdr( FBlkHeader &h, long p)
106 // Writes out free block header.
107 {
108     store(&h, sizeof(FBlkHeader), p);
109 }
110
111 /*
112 inline long fmgr::Aux(int i) const
113 {
114     if ( i >= AUXHEADERS ) {
115         throw VcfileError( "fmgr::Aux", "Access beyond vector" );
116     }
117     return h.aux[i];
118 }
119 */
120 inline long &fmgr::Aux(int i)
```

```
 1 /*                                                                    1
 2  EPSHeader                                                    96STTE"
 3
 4     File: filmatch.c
 5     Author: J. Kercheval
 6     Created: Thu, 03/14/1991  22:22:01
 7 */
 8
 9 /*
10  EPSRevision History
11
12     J. Kercheval  Wed, 02/20/1991  22:29:01  Released to Public Domain
13     J. Kercheval  Fri, 02/22/1991  15:29:01  fix '\' bugs (two :( of them)
14     J. Kercheval  Sun, 03/10/1991  19:31:29  add error return to matche()
15     J. Kercheval  Sun, 03/10/1991  20:11:11  add is_valid_pattern code
16     J. Kercheval  Sun, 03/10/1991  20:37:11  beef up main()
17     J. Kercheval  Tue, 03/12/1991  22:25:10  Released as V1.1 to Public Domain
18     J. Kercheval  Thu, 03/14/1991  22:22:25  remove '\' for DOS file parsing
19     J. Kercheval  Thu, 03/28/1991  20:58:27  include filmatch.h
20 */
21
22 /*
23     Wildcard Pattern Matching
24 */
25
26
27 #include "match.h"
28
29 int matche_after_star (char *pattern, char *text);
30 int fast_match_after_star (char *pattern, char *text);
31
32 /*--------------------------------------------------------------------
33 *
34 * Return MATCH_TRUE if PATTERN has any special wildcard characters
35 *
36 --------------------------------------------------------------------*/
37
38 BOOLEAN is_pattern (char *p)
39 {
40     while ( *p ) {
41         switch ( *p++ ) {
42             case '?':
43             case '*':
44             case '[':
45                 return MATCH_TRUE;
46         }
47     }
48     return MATCH_FALSE;
49 }
50
51
52 /*--------------------------------------------------------------------
53 *
54 * Return MATCH_TRUE if PATTERN has is a well formed regular expression according
55 * to the above syntax
56 *
57 * error_type is a return code based on the type of pattern error.  Zero is
58 * returned in error_type if the pattern is a valid one.  error_type return
59 * values are as follows:
60 *
_squibm_____C:\TMP\VI\MATCH.CPP_____Sat Jul 20 14:59:38 1996_

                                                                       2
61 *    PATTERN_VALID - pattern is well formed
62 *    PATTERN_ESC - pattern has a trailing escape (bad pattern)
63 *    PATTERN_RANGE - [..] construct has a no end range in a '-' pair (ie [a-])
64 *    PATTERN_CLOSE - [..] construct has no end bracket (ie [abc-g )
65 *    PATTERN_EMPTY - [..] construct is empty (ie [])
66 *
67 --------------------------------------------------------------------*/
68 BOOLEAN is_valid_pattern (char *p, int *error_type)
69 {
70
71     /* init error_type */
72     *error_type = PATTERN_VALID;
73
74     /* loop through pattern to EOS */
75     while( *p ) {
76
77         /* determine pattern type */
78         switch( *p ) {
79
80             /* the [..] construct must be well formed */
81             case '[':
82                 p++;
83
84                 /* if the next character is ']' then bad pattern */
85                 if ( *p == ']' ) {
86                     *error_type = PATTERN_EMPTY;
87                     return MATCH_FALSE;
88                 }
89
90                 /* if end of pattern here then bad pattern */
91                 if ( !*p ) {
92                     *error_type = PATTERN_CLOSE;
93                     return MATCH_FALSE;
94                 }
95
96                 /* loop to end of [..] construct */
97                 while( *p != ']' ) {
98
99                     /* check for literal escape */
100                    if( *p == '\\' ) {
101                        p++;
102
103                        /* if end of pattern here then bad pattern */
104                        if ( !*p++ ) {
105                            *error_type = PATTERN_ESC;
106                            return MATCH_FALSE;
107                        }
108                    }
109                    else
110                        p++;
111
112                    /* if end of pattern here then bad pattern */
113                    if ( !*p ) {
114                        *error_type = PATTERN_CLOSE;
115                        return MATCH_FALSE;
116                    }
117
118                    /* if this a range */
119                    if( *p == '-' ) {
120
```

```
121                     /* we must have an end of range */
122                     if ( !*++p || *p == ']' ) {
123                         *error_type = PATTERN_RANGE;
124                         return MATCH_FALSE;
125                     }
126                     else {
127
128                         /* check for literal escape */
129                         if( *p == '\\' )
130                             p++;
131
132                         /* if end of pattern here then bad pattern */
133                         if ( !*p++ ) {
134                             *error_type = PATTERN_ESC;
135                             return MATCH_FALSE;
136                         }
137                     }
138                 }
139             }
140             break;
141
142         /* all other characters are valid pattern elements */
143         case '*':
144         case '?':
145         default:
146             p++;                            /* "normal" character */
147             break;
148         }
149     }
150
151     return MATCH_TRUE;
152 }
153
154
155 /*----------------------------------------------------------------------
156  *
157  * Match the pattern PATTERN against the string TEXT;
158  *
159  * returns MATCH_VALID if pattern matches, or an errorcode as follows
160  * otherwise:
161  *
162  *          MATCH_PATTERN   - bad pattern
163  *          MATCH_RANGE     - match failure on [..] construct
164  *          MATCH_ABORT     - premature end of text string
165  *          MATCH_END       - premature end of pattern string
166  *          MATCH_VALID     - valid match
167  *
168  *
169  * A match means the entire string TEXT is used up in matching.
170  *
171  * In the pattern string:
172  *      '*' matches any sequence of characters (zero or more)
173  *      '?' matches any character
174  *      [SET] matches any character in the specified set,
175  *      [!SET] or [^SET] matches any character not in the specified set.
176  *      \ is allowed within a set to escape a character like ']' or '-'
177  *
178  * A set is composed of characters or ranges; a range looks like
179  * character hyphen character (as in 0-9 or A-Z).  [0-9a-zA-Z_] is the
180  * minimal set of characters allowed in the [..] pattern construct.
```



```
181  * Other characters are allowed (ie. 8 bit characters) if your system
182  * will support them.
183  *
184  * To suppress the special syntactic significance of any of '[]*?!^-\',
185  * within a [..] construct and match the character exactly, precede it
186  * with a '\'.
187  *
188  *----------------------------------------------------------------------*/
189
190 int matche ( char *p, char *t )
191 {
192     register char range_start, range_end;  /* start and end in range */
193
194     BOOLEAN invert;             /* is this [..] or [!..] */
195     BOOLEAN member_match;       /* have I matched the [..] construct? */
196     BOOLEAN loop;               /* should I terminate? */
197
198     for ( ; *p; p++, t++ ) {
199
200         /* if this is the end of the text then this is the end of the match */
201         if (!*t) {
202             return ( *p == '*' && *++p == '\0' ) ? MATCH_VALID : MATCH_ABORT;
203         }
204
205         /* determine and react to pattern type */
206         switch ( *p ) {
207
208             /* single any character match */
209             case '?':
210                 break;
211
212             /* multiple any character match */
213             case '*':
214                 return matche_after_star (p, t);
215
216             /* [..] construct, single member/exclusion character match */
217             case '[': {
218
219                 /* move to beginning of range */
220                 p++;
221
222                 /* check if this is a member match or exclusion match */
223                 invert = MATCH_FALSE;
224                 if ( *p == '!' || *p == '^' ) {
225                     invert = MATCH_TRUE;
226                     p++;
227                 }
228
229                 /* if closing bracket here or at range start then we have a
230                    malformed pattern */
231                 if ( *p == ']' ) {
232                     return MATCH_PATTERN;
233                 }
234
235                 member_match = MATCH_FALSE;
236                 loop = MATCH_TRUE;
237
238                 while ( loop ) {
239
240                     /* if end of construct then loop is done */
```

```
241                 if (*p == ']') {
242                     loop = MATCH_FALSE;
243                     continue;
244                 }
245
246                 /* matching a ']', '^', '-', '\' or a ']' */
247                 if ( *p == '\\' ) {
248                     range_start = range_end = *++p;
249                 }
250                 else {
251                     range_start = range_end = *p;
252                 }
253
254                 /* if end of pattern then bad pattern (Missing ']') */
255                 if (!*p)
256                     return MATCH_PATTERN;
257
258                 /* check for range bar */
259                 if (*++p == '-') {
260
261                     /* get the range end */
262                     range_end = *++p;
263
264                     /* if end of pattern or construct then bad pattern */
265                     if (range_end == '\0' || range_end == ']')
266                         return MATCH_PATTERN;
267
268                     /* special character range end */
269                     if (range_end == '\\') {
270                         range_end = *++p;
271
272                         /* if end of text then we have a bad pattern */
273                         if (!range_end)
274                             return MATCH_PATTERN;
275                     }
276
277                     /* move just beyond this range */
278                     p++;
279                 }
280
281                 /* if the text character is in range then match found.
282                    make sure the range letters have the proper
283                    relationship to one another before comparison */
284                 if ( range_start < range_end ) {
285                     if (*t >= range_start && *t <= range_end) {
286                         member_match = MATCH_TRUE;
287                         loop = MATCH_FALSE;
288                     }
289                 }
290                 else {
291                     if (*t >= range_end && *t <= range_start) {
292                         member_match = MATCH_TRUE;
293                         loop = MATCH_FALSE;
294                     }
295                 }
296             }
297
298             /* if there was a match in an exclusion set then no match */
299             /* if there was no match in a member set then no match */
300             if ((!invert && member_match) ||
```

```
301                 !(invert || member_match))
302                     return MATCH_RANGE;
303
304                 /* if this is not an exclusion then skip the rest of the [...]
305                    construct that already matched. */
306                 if (member_match) {
307                     while (*p != ']') {
308
309                         /* bad pattern (Missing ']') */
310                         if (!*p)
311                             return MATCH_PATTERN;
312
313                         /* skip exact match */
314                         if (*p == '\\') {
315                             p++;
316
317                             /* if end of text then we have a bad pattern */
318                             if (!*p)
319                                 return MATCH_PATTERN;
320                         }
321
322                         /* move to next pattern char */
323                         p++;
324                     }
325                 }
326
327                 break;
328             }
329
330             /* must match this character exactly */
331             default:
332                 if (*p != *t)
333                     return MATCH_LITERAL;
334         }
335     }
336
337     /* if end of text not reached then the pattern fails */
338     if ( *t )
339         return MATCH_END;
340     else
341         return MATCH_VALID;
342 }
343
344
345 /*----------------------------------------------------------------------
346  *
347  * recursively call matche() with final segment of PATTERN and of TEXT.
348  *
349  ----------------------------------------------------------------------*/
350
351 int matche_after_star ( char *p,  char *t)
352 {
353     register int match = 0;
354     register nextp;
355
356     /* pass over existing ? and * in pattern */
357     while ( *p == '?' || *p == '*' ) {
358
359         /* take one char for each ? and * */
360         if ( *p == '?' ) {
```

```
361
362             /* If end of text then no match */
363             if ( !*t++ ) {
364                 return MATCH_ABORT;
365             }
366         }
367
368         /* move to next char in pattern */
369         p++;
370     }
371
372     /* if end of pattern we have matched regardless of text left */
373     if ( !*p ) {
374         return MATCH_VALID;
375     }
376
377     /* get the next character to match which must be a literal or '[' */
378     nextp = *p;
379
380     /* Continue until we run out of text or definite result seen */
381     do {
382
383         /* a precondition for matching is that the next character
384            in the pattern match the next character in the text or that
385            the next pattern char is the beginning of a range.  Increment
386            text pointer as we go here */
387         if ( nextp == *t || nextp == '[' ) {
388             match = matche(p, t);
389         }
390
391         /* if the end of text is reached then no match */
392         if ( !*t++ ) match = MATCH_ABORT;
393
394     } while ( match != MATCH_VALID &&
395               match != MATCH_ABORT &&
396               match != MATCH_PATTERN);
397
398     /* return result */
399     return match;
400 }
401
402
403 /*----------------------------------------------------------------------
404  *
405  * match() is a shell to matche() to return only BOOLEAN values.
406  *
407  *--------------------------------------------------------------------*/
408
409 BOOLEAN match( char *p, char *t )
410 {
411     int error_type;
412     error_type = matche(p,t);
413     return (error_type == MATCH_VALID ) ? MATCH_TRUE : MATCH_FALSE;
414 }
415
416
417 #ifdef TEST
418
419     /*
420      * This test main expects as first arg the pattern and as second arg
```



```
421      * the match string.  Output is yeah or nay on match.  If nay on
422      * match then the error code is parsed and written.
423      */
424
425     #include <stdio.h>
426
427     int main(int argc, char *argv[])
428     {
429         int error;
430         int is_valid_error;
431
432         if (argc != 3) {
433             printf("Usage:  MATCH Pattern Text\n");
434         }
435         else {
436             printf("Pattern: %s\n", argv[1]);
437             printf("Text   : %s\n", argv[2]);
438
439             if (!is_pattern(argv[1])) {
440                 printf("    First Argument is Not A Pattern\n");
441             }
442             else {
443                 match(argv[1],argv[2]) ? printf("MATCH_TRUE") : printf("MATCH_FA
    LSE");
444                 error = matche(argv[1],argv[2]);
445                 is_valid_pattern(argv[1],&is_valid_error);
446
447                 switch ( error ) {
448                     case MATCH_VALID:
449                         printf("    Match Successful");
450                         if (is_valid_error != PATTERN_VALID)
451                             printf(" -- is_valid_pattern() is complaining\n");
452                         else
453                             printf("\n");
454                         break;
455                     case MATCH_RANGE:
456                         printf("    Match Failed on [..]\n");
457                         break;
458                     case MATCH_ABORT:
459                         printf("    Match Failed on Early Text Termination\n");
460                         break;
461                     case MATCH_END:
462                         printf("    Match Failed on Early Pattern Termination\n"
    );
463                         break;
464                     case MATCH_PATTERN:
465                         switch ( is_valid_error ) {
466                             case PATTERN_VALID:
467                                 printf("    Internal Disagreement On Pattern\n");
    }
468                                 break;
469                             case PATTERN_RANGE:
470                                 printf("    No End of Range in [..] Construct\n"
    );
471                                 break;
472                             case PATTERN_CLOSE:
473                                 printf("    [..] Construct is Open\n");
474                                 break;
475                             case PATTERN_EMPTY:
476                                 printf("    [..] Construct is Empty\n");
```

```
/* Copyright 1995 Mark Squibb */

#include <mgr_file.h>
#include <sskglob.h>
#include <fcntl.h>
#include <io.h>
#include <sys\stat.h>
#include <process.h>

//////////////////////////////////////////////////////////////////////////
//////////// Member Functions for:  File_Mgr   (mgr_file.cpp)  ////////////
//
// Possible Errors
//  None
void File_Mgr::SetName (char* name)
{
  FileName = name;
}


File_Mgr::File_Mgr(void){
    FD     = 0;                  // File descriptor for working Table File
    BaseFileSize = 0;            // Current size of Table File
    here = 0;                    // Current positon within Table File
    Extended = OFF;              // Flag for file extension
    Handle = 0;
    EvnJrn = NULL;                   // pointer to RollbackJournal
    FileActiveCount = 0;
    ObjectNotRestorableFlag = 0;
}


File_Mgr::~File_Mgr(void){
  Close();
  FileActiveCount = 0;
}


// Creates the Table File and sets it's file descriptor: FD
//
// Possible Errors:
//  1) Open fails on file error
//
void File_Mgr::Create( )
{
    //UNIX: FD = open((const char *)FileName, O_CREAT | O_TRUNC );
  FD = open((const char *)FileName,
          O_CREAT | O_TRUNC | O_RDWR | O_BINARY,
          S_IREAD | S_IWRITE );
  if( FD < 0 ) {
    FileError FE( "File_Mgr::Create", FileName  );
    FE.SetErrCode( FILE_CREATE_ERROR );
    throw FE;
  }
  // Test to make sure file opened in binary mode
  // I've had bunches of problems with this, so test whenever a
  // file is opened
#if defined ( __BORLANDC__ ) || defined( WIN32 )
  write( FD, "\n", 1 );

  // Win95 is wacked.  Stat doesn't return the correct size!
  // Try the old dos trick to get up to date stat info
  int DupFd = dup( FD );
  close( DupFd );

  struct stat sb;
  fstat( FD, &sb );
  if ( sb.st_size != 1 ) {
    throw ProgramError( "File_Mgr::Create", "File opened in text mode" );
  }
  // Microsoft doesn't reset the file position with truncate!
  lseek( FD, 0, SEEK_SET );
  chsize( FD, 0 );
#endif
#if defined( _FILE_MGR_DEBUG_ )
  cout << "Opened:       " << FileName << endl;
  cout << "Descriptor:  " << FD << endl;
#endif;
   here=0L;
}

// Opens the Table File for read/write operations
//
// Possible Errors:
//  1) Open fails on file error
//
void File_Mgr::Open( )
{
    FD = open( FileName.Ptr(), O_RDWR | O_BINARY,
               S_IREAD | S_IWRITE );

    if( FD < 0 ) {
    FileError FE( "File_Mgr::Open", FileName );
    FE.SetErrCode( FILE_OPEN_ERROR );
    throw FE;
  }

  #if defined( _FILE_MGR_DEBUG_ )
  cout << "Opened:       " << FileName << endl;
  cout << "Descriptor:  " << FD << endl;
  #endif;

  BaseFileSize = Seek( 0, SEEK_END );
  Seek( 0, SEEK_SET );
}

// Closes Table File and sets 'here' to -1
//
// Possible Error
//  1)  Close of file (Unlikely)

void File_Mgr::Close(void){
  if ( EvnJrn )
    DetachJournal();

  if ( FD > 0 ) close( FD );
  FD = -1;
  here = -1;

  // This is more of a warning than an error
```


000067

```
121   // but the user should be notified if there is an active
122   // user during closure to help diagnose reading errors
123   // attached to no file
124   if ( FileActiveCount ) {
125     throw ProgramError( "File_Mgr::Close",
126                         "File was active at close" );
127   }
128
129 }
130
131 // Returns the current position in the Table file:  'here'
132 //
133 // Possible Errors:
134 //   None
135 long File_Mgr::Tell(void)
136 {
137   return( here );
138 }
139
140 // Truncates the Table File to a specified length
141 //
142 void File_Mgr::Chsize( long SZ){
143   chsize( FD, SZ );
144   Seek( 0, SEEK_END );
145   BaseFileSize = SZ;
146 }
147
148 // Seeks to a position within Table File and sets 'here'
149 // the force parameter forces the file to seek to the defined position
150 // because the file may have been modified outside the mgr_file in rbjrn
151 //
152 //  Possible Errors:
153 //    1)  lseek failure
154 //
155 long File_Mgr::Seek( long where, int whence, int ForceSeek )  {
156   // Check to see if we're writing anyplace other than 'here'
157   int seekflag = ON;
158   switch ( whence ) {
159   case SEEK_SET:
160     if ( here == where ) seekflag = OFF;
161     break;
162   case SEEK_CUR:
163     if ( 0 == where ) seekflag = OFF;
164     break;
165   }
166   if ( ForceSeek ) seekflag = ON;
167
168   if ( seekflag ){
169     where = lseek( FD, where, whence );
170     if ( where < 0 ) {
171       FileError FE( "File_Mgr::Seek", FileName ) ;
172       FE.SetErrCode( FILE_SEEK_ERROR );
173       throw FE;
174     }
175     here = where;
176   }
177   return(here);
178 }
179
180 // Read buffer from Table File
```



```
181 //
182 // Possible Error
183 //   1) Physical device error on read
184 int File_Mgr::Read( void *buf, int Get ) {
185   int Got = read( FD, (char*)buf, Get );
186   if ( Got >= 0 ) here += Got;
187   else {
188     FileError FE( "File_Mgr::Read", FileName );
189     FE.SetErrCode( FILE_READ_ERROR );
190     throw FE;
191   }
192   return Got;
193 }
194
195 // Log event transaction in Rollback File and write buffer to Table File
196 //
197 // Possible Errors
198 //  1) Physical Write Error
199
200 int File_Mgr::Write( void *buf, int cnt )
201 {
202   if ( EvnJrn ) {  // If a transaction is open, journal the event
203     JournalEvent ( cnt );
204   }
205   int Put = write( FD, (char*)buf, cnt );
206   if ( Put < 0 ) {
207     FileError FE( "File_Mgr::Write", FileName );
208     FE.SetErrCode( FILE_WRITE_ERROR );
209     throw FE;
210   }
211   if ( Put > 0 ) { here += Put; }
212   return Put;
213 }
214
215
216 ///JournalEvent////////////////////////////////////////////////////////////
217 //
218 // Logs a write transaction to a Table File.
219 //
220 int File_Mgr::JournalEvent( int Cnt )
221 {
222   EventDescripter ET;
223
224   if ( here >= BaseFileSize ) {
225     // extend transaction
226     if ( !Extended ) { // no need to add more ex
227       ET.Handle   = Handle;
228       ET.Type     = LIBFILE_EVENT_EXTEND;
229       // ET.Size     = sizeof( long );
230       ET.Position = BaseFileSize;
231
232       EvnJrn->StartEvent();
233       EvnJrn->SetEventDescripter( ET );
234       EvnJrn->EndEvent();
235       Extended = ON;
236     }
237
238   } else {    // Read the area about to be overtyped
239     // Initialize a new event
240     EvnJrn->StartEvent();
```

```
261
262     // Read the overtypo data                                    965777
263     long WriteTarget = here;        // Save the write target
264     if ( (int)ReadBuf.size() < Cnt ) {
265       ReadBuf.resize( Cnt );
266     }
267     // Bufr ReadBuf( Cnt );             // Allocate a read buffer
268
269     //static char* HoldPtr;
270     //HoldPtr = ReadBuf.Ptr();
271     Read( ReadBuf.Ptr(), Cnt );     // Load data to be overwritten
272
273     // Transmit the data to the event journal
274     EvnJrn->SetEventData( ReadBuf.Ptr(), Cnt );
275
276     // Issue an event marker to the Rollback file
277     ET.Handle    = Handle;
278     ET.Type      = LIBFILE_EVENT_OVERTYPE;
279     // ET.Size      = Cnt;
280     ET.Position  = WriteTarget;
281     EvnJrn->SetEventDescriptor( ET );
282     EvnJrn->EndEvent();
283
284     // Go back to the target location
285     Seek( WriteTarget );
286     //if ( HoldPtr != ReadBuf.Ptr() ) {
287     //  cerr << "Pointer changed during execution" << endl;
288     //  cin.getline( "dummystring", 10 );
289     //}
290   }
291   return 0;
292 }
293
294 // If a transaction extends the file
295 //  then the new location checks BaseFileSize
296 //  to see if the file was extended
297 //
298 //  ResetExtended, resets the base file size
299 //    and turns off the extended flag
300
301 void File_Mgr::ResetExtended ( )
302 {
303   Extended = Off;
304   long HereSave = here;
305   if ( FD == -1 ) return;
306   BaseFileSize = Seek( 0, SEEK_END );
307   Seek( HereSave, SEEK_SET );
308 }
309 /*
310 int File_Mgr::GetHandle ( )
311 {
312   return Handle;
313 }
314 */
315 int File_Mgr::SetHandle ( int hdl )
316 {
317     Handle = hdl;
318     return Handle;
319 }
320
```



```
321 int File_Mgr::GetFD ( )                                    0000069
322   return FD;
323 }
324
325 //
326 // Attach the file manager to an event journal
327 //  1) Set the local pointer
328 //  2) Advance to the end of the journal pointer collection
329 //  3) Insert a pointer to this instance
330 //  4) Set this handle to the appropriate list index
331 //
332 //  Possible Errors:
333 //    Event Journal has uncommitted transactions
334 //
335
336 void File_Mgr::AttachJournal ( RollbackJournal *EvJn ){
337   // Check the event journal for uncommitted transactions
338   if ( EvJrn->CheckCommitted() == 0 )
339     throw( ProgramError( "File_Mgr::AttachJournal",
340                          "Must Commit before attaching any files" )
341     );
342   // Set the extension variable to the current file size
343   // Flush any users of the table at this point to ensure
344   // any cached data is written to disk
345   if ( FD > 0 ) {
346     ResetExtended();
347     FlushAllUsers();
348   }
349   RawAttachJournal( EvJn );
350   // EvJrn->FlushHeader();
351 }
352
353 //
354 // Drop this file manager from the event journal
355 //
356 //  1)  Look up the appropriate node
357 //  2)  Null out the pointer to this object
358 //  3)  Null the EvnJrn pointer
359 //
360 void File_Mgr::DetachJournal( ){
361   // Select the node
362   // EvnJrn->SelectFmgr( Handle );
363   // Drop the node from the event manager
364   // EvnJrn->delX();
365
366   EvnJrn->DropFileMgr( Handle );
367   // Break the pointer to the journaler
368   EvnJrn = NULL;
369 }
370 //
371 // During crash recovery, the Event Journal has uncommitted
372 //  transactions.  This function allows attachment under
373 //  these conditions.
374 //
375 void File_Mgr::RawAttachJournal( RollbackJournal *EvJn ) {
376
377   EvnJrn = EvJn;
378   EvnJrn->AttachFmgr( this );
379 }
```

000070
S00°00'09"E 1115.96
361
C:\TMP\VTWGB_FILE.CPP

```
1 /* Copyright 1995 Mark Squibb */
2
3 ////////////////////////////////////////////////////////////////////////
4 ////////// Class definition for: File Manager  (mgr_file.hpp) //////////
5 ////////////////////////////////////////////////////////////////////////
6
7 #ifndef _LIB_FILE_HPP_
8 #define _LIB_FILE_HPP_
9 #include <cl.h>
10
11 #include <io.h>
12 #include <stdio.h>
13 #include <fcntl.h>
14 #include <buf.h>
15 #include <exception.h>
16 #include <asklgbl.h>
17
18 #include <iomanip.h>
19 ////////////////////////////////////////////////////////////////////////
20 //
21 //  The FILE MANAGER
22 //
23 //  The File Manager performs the basic io operations used to update the
24 //  Table File (i.e. Create, Open, Close, Read, and Write).
25 //
26 ////////////////////////////////////////////////////////////////////////
27
28
29 class RollbackJournal;
30
31 class EXPORTSPEC File_Mgr {
32
33 protected:
34
35   int  FD;                  // file descriptor of working Table File
36   Buf  FileName;            // Name of working Table File
37   long BaseFileSize;        // Current size of Table File
38   long here;                // Current position in Table File
39   int  Extended;            // Has the transaction extended the file
40   RollbackJournal *EvnJrn;  // pointer to RollbackJournal class
41
42   Buf* ReadBuf;             // Buffer for rollback reads
43
44   int  JournalEvent ( int Cnt );
45   void AssAttachJournal( RollbackJournal* EJ );
46
47   // this flag enables using classes to tell the file manager class
48   // when they are actively using the file.  If the application attempts
49   // to rollback while a process is active, the pending operation throws
50   // an error.
51   int      fileActiveCount;
52
53   // This flag enables the application to say when it is in a non-committable
54   // state.  For example if the instance crashed, the physical file could
55   // not be used to reconstruct the object instances which are stored in the f
56   int      ObjectNotRestorableFlag;
57
58   friend RollbackJournal;
59   int  GetFD ();
60   void VoidLogEXgraded ( );
61   int  IsInSandbox ( int Hdl );
62   int  Handle;              // Must be set by Table Manager or EventJournal
63
64   // Before a commit all the users of an instance must be caused to write
65   // any cached data to disk to ensure the disk file correctly represents
66   // the data.  Direct users have no cache so the default
67   // version does nothing
68   virtual void FlushAllUsers();
69   // When a rollback occurs, the file data for an object may change
70   // leaving cached data out of synch with disk.  The ShutDown users
71   // is called for each file manager to terminate each active file user
72   // before the rollback occurs
73   virtual void ShutDownUsers();
74   // After a rollback, the user objects must be reestablished from the disk
75   // the restart function reloads the user processes from the disk
76   virtual void RestartUsers();
77
78 public:
79
80   File_Mgr ();
81   ~File_Mgr (void);
82   void SetName (char* name);
83   char* GetName() { return FileName.Ptr(); }
84
85   void Create ( );              // Creates Table File and set 'FD'
86   void Open ( );                // Opens Table File for read/write
87   void Close (void);            // Closes Table File and sets 'here' to -1
88   long Tell ( void );           // Returns current pos (here) in Table File
89   void Chsize (long);           // Truncates Table and resets 'BaseFileSize'
90   long Seek (long where=0, int whence=SEEK_SET, int Force = OFF );
91   int  Read (void *, int);      // Read buffer from Table File
92   int  Write (void *, int);     // Log event in Rollback and write to Table
93
94   void AttachJournal ( RollbackJournal *Evjn );
95   void DetachJournal( );
96
97   // These functions indicate whether the object set is restorable from the
98   // physical file image
99   virtual void SetObjectNotRestorable() { ObjectNotRestorableFlag = 0; }
100  virtual void SetObjectNotRestorable() { ObjectNotRestorableFlag = 1; }
101
102  // If a user object is actively using the file, than
103  // The process should notify the file_mgr to prevent
104  // a rollback from trashing the current process
105  virtual void SetFileActive() { fileActiveCount++; }
106  virtual void SetFileInactive() { if ( fileActiveCount ) fileActiveCount--; }
107  };
108
109  // Returns a count of how many times the file is active
110  virtual int IsFileActive() { return fileActiveCount; }
111
112  // Returns whether the dependent objects are restorable from file image
113  virtual int IsObjectRestorable() { return !ObjectNotRestorableFlag; };
114
115
116 ////////////////////////////////////////////////////////////////////////////
117 //
118 //  EVENT JOURNAL CLASS
```

```
119 //
120 //  The Event Journal logs all write transactions (events) to the
121 //  Table File(s) in a special file called the Rollback file.  Two
122 //  types of events are tracked:  1) Extend and 2) Overtype.  Extend
123 //  transactions record instances where data is simply added to the
124 //  end of the Table file (without modify existing data).  Overtype
125 //  transactions, on the other hand, record instances where existing
126 //  data within the Table File has been modified.  As a result, it is
127 //  necessary to save the existing data in the Table file (which is
128 //  then stored in the Rollback file) before an overwrite operation is
129 //  performed.
130 //
131 //  In general, the Rollback file works as follows when undoing changes
132 //  made to the Table files:
133 //
134 //      1)  the rollback must be played backwards to achieve the Table
135 //          file's base state (i.e. each record in the Rollback file ends
136 //          with an Event Trailer that points BACK to the previous record
137 //          (event transaction) in the file)
138 //
139 //      2)  As the transactions are iterated through, the Extend and
140 //          Overtype Transaction are evaluated as follows:
141 //
142 //              a) When an Extend transactions is encountered, the Table
143 //                  file is simply truncated at that point.
144 //
145 //              b) When a Overtype Transaction is encountered, the original
146 //                  data is read from the Rollback file, which is then written
147 //                  back to the Table file at the offset that was saved.
148 //
149 //  Also... this facility does not attempt to protect data from truncation
150 //      if a subject file is truncated, all of the subsequent data is lost.
151 //      The appropriate rule would seem to be that any action requiring a
152 //      chsize (table collapse) would implicitly commit.
153 //
154 //  ISSUE:  If a Table file is rolled back to a previous state, how is
155 //  the Table's Index File updated (i.e. rolled back)
156 //
157 ////////////////////////////////////////////////////////////////////////
158
159 // Event transaction types:
160
161 #define LIBFILE_EVENT_OVERTYPE  ( 1 )
162 #define LIBFILE_EVENT_EXTEND    ( 2 )
163 #define LIBFILE_EVENT_SAVEPOINT ( 3 )
164 #define LIBFILE_EVENT_HEADER    ( 4 )
165
166 /*
167  Event structure definition
168 */
169 struct EventDescriptor {  // forms a record in the Rollback file
170     long Handle;  // Table File Handle
171     long Type;    // Event transaction type
172     long Size;    // Size of data
173     long Position; // offset position of data within Table File
174     void Clear() { Handle=0; Type=0; Size=0; Position=0; }
175     EventDescriptor() { Clear(); }
176     void print(ostream& os ) {
177         os << setw( 10 ) << Handle
178            << setw( 10 ) << Type
                                                    C:\INDEX\VTV\MCA_FILE.H
179            << setw( 10 ) << Size
180            << setw( 12 ) << Position
181            << endl;
182     }
183 };
184
185 #define MAX_EVENT_SIZE ( 10240 )
186
187 // Unstreamable and uncomparable
188 CL_NO_stream( File_Mgr );
189 CL_NO_copy( File_Mgr );
190 CL_NO_assign( File_Mgr );
191
192 //class TransactionServices;
193
194 class RollbackJournal : public CL <File_Mgr>
195 {
196     protected:
197
198         int     Rib;            // File descriptor of Rollback file
199         long    here;           // offset in current file
200         Sbuf    RbName;         // Name of Rollback file
201
202         int     CommittedFlag;  // Flag indicating object is committed
203
204         int     NextHandle;     // Counter to assign next handle
205
206         Buff    EventDataBuffer;  // buffer to hold event data
207         int     EventDataSize;    // How much valid data is in the event data buffer
208         EventDescriptor Event;    // Descriptor for current event
209
210         long ActiveSavePoint;   // Counter for savepoint
211
212         void RefreshHeaderSized );
213         void DestroyTempFiles ( );          // Reset the size param of header
214         int  RollbackExist();               // Destroy's temp file Managers
215                                             // Does Rollback file exist
216         void FlushHeader();                 // Write out the header segment
217         void ReadHeader ( int IgnoreMissingFlag );
218         void Chsize( long SZ);
219         long Seek( long Where, int Whence );
220         int  Read( void *buf, int Cat );
221         int  Write( void *buf, int Cat );
222         void Open( );
223         int  SelectFmgr ( int Hdl );
224         void SetName( char* DatabaseName );  // Make 'Hdl' the current Fmgr
225         void Create( );                      // Set the Rollback file name
226
227
228 friend File_Mgr;
229 //-Event recording commands
230 void StartEvent();   // Reset the event process
231 void SetEventData( void* EventData, int Cnt ); // Copy data portion of event
232 void SetEventDescriptor( EventDescriptor& ED ); // Copy event descriptor
233 void EndEvent();     // Write event to event file public:
234 void AttachFmgr ( File_Mgr* Fmgr );  // Attach File Mgr to this object
235 int  GetNextHandle() {
236     int RetVal = NextHandle; NextHandle++; return RetVal;
237 }
238 long LoadEvent( long EventLocation );
                                                    Mon Aug 26 11:29:16 1996_
```

```
239   void DropFileMgr( int Handle );
240
241
242   // TransactionServiceBase
243   // Pointer to aux service class which gets called
244   // when a significant event occurs
245   //TransactionServiceBase* TSB;
246
247  public:
248   RollbackJournal (void);
249   ~RollbackJournal (void);
250   void  Close(void);
251
252   // Function creates a savepoint in the transaction log enabling
253   // a rollback to stop at this position instead of reverting to
254   // the last committed state.
255   long  CreateSavePoint();
256
257   void Commit ();                          // Make changes to files permanent
258   void Rollback ( long SavePoint = 0 );      // Revert all transactions
259
260   void StartRollback (char * DbName,
261                       int SalvageFlag ); // Create and open Rollback file
262
263   void ShutDown();          //  Finish existence of rollback object
264   int  CheckCommitted();    //  Returns OK if committed
265 );
266
267
268 /* EventJournalError
269    Describes logical errors with event journal mechanism
270 */
271
272 // Errors for event journal
273 #define RollbackJournal_CORRUPT   ( 1 )
274
275 class EXPORTSPEC RollbackJournalError : public Exception {
276  public:
277   RollbackJournalError( char* Funct ) : Exception( Funct ) {
278     ExceptionClassCode = RollbackJournal_EXCEPTION_CODE;
279   };
280
281   void Report() {
282     cerr << "RollbackJournalError thrown" << endl;
283     Exception::Report();
284   }
285
286 };
287
288 #define FILEMGR_EXCEPTION_CODE ( 3 )
289 class EXPORTSPEC FileMgrError : public Exception {
290  protected:
291   char* LogicalErr;
292  public:
293   FileMgrError( char* Funct, char* msg1= NULL ) : Exception( Funct ) {
294     LogicalErr = msg1;
295     ExceptionClassCode = FILEMGR_EXCEPTION_CODE;
296   };
297
298   virtual char* GetReportString() {
```



```
299     static char ReportString[ 512 ];
300     memset( ReportString, 0, 512 );
301
302     strcpy( ReportString, "FileMgrError Thrown\n" );
303
304     if ( strlen( FunctionName ) ) {
305       strcpy( ReportString, "Function Name:  " );
306       strcat( ReportString, "  " );
307       strcat( ReportString, FunctionName );
308       strcat( ReportString, "\n" );
309     }
310
311     if ( strlen( LogicalErr ) ){
312      strcat( ReportString, LogicalErr );
313      strcat( ReportString, "\n" );
314    }
315    return ReportString;
316  }
317 };
318 #endif  // _LIBR_FILE_HPP_
```

```
1 ////////////////////////////////////////////////////////////////////////////
2 // placement.hi inline placement new operator                    95111 "960803
3 // Copyright(c) 1993 Azarona Software. All rights reserved.
4 ////////////////////////////////////////////////////////////////////////////
5 #ifndef M_PLACEMEM
6 #define M_PLACEMEM
7 #include <stddef.h>
8
9 // NOTE: If you are using Symantec C++ for the Mac, this
10 // operator is already defined in stdlib.h or stddef.h,
11 // (I don't know which, I got this information second hand.)
12
13 inline void *operator new(size_t, void *p)
14 // Placement new operator.
15 {
16   return p;
17 }
18
19 #endif
20
```

```
 1 /*  Copyright 1995 Mark Squibb                                    96STTT
 2   The purpose of this object:                                    96STTT
 3     to provide state integrity for files maintained by File_Mgr objects
 4
 5     In a reversion context...
 6     Writes to existing data space may overtype existing data.  As such
 7     lifting the data to be overtyped and recording it in a journal such
 8     as this enables the overtype process to be reversed.
 9
10   General Services:
11
12     Return file to consistent state at startup
13       When a program aborts, it may leave its working files in an
14       inconsistent state.  This object in conjunction with the file
15       manager automatically returns the files that were connected
16       at the time of the abortion to the last committed state.
17
18     Provide Reversion Services to application
19       An application may encounter circumstances where it is desirable
20       to abandon a series of changes which were made.  This object
21       enables the application to select important operating instances
22       and to reverse activity since those instances.  The Commit and
23       Rollback functions enable these services.
24
25     Provide Savepoint Services to an application
26       There may be circumstances where a series of operations are
27       related by more complex contingencies.  Savepoints enable
28       the application to define lesser commit points.  A lesser commit
29       point effectively enables the application to abandon all of the
30       activity since a particular instant without abandoning all
31       of the activity in a session.
32
33     Synchronization Services:
34       Applications may contain information in memory.  As such the information
35       on disk and in memory may be out of synch.  At key points, specifically
36       when a retrievable instance is required, information in disk and in memory
37       must be synchronized.  At these key points, the event journal calls the
38       FlushAllUsers method for each attach file manager to cause the file manage
rs
39       application to write any data in memory to disk.  (Retrievable instances
40       refer to commit and savepoints).
41
42       Rollbacks invariably change file information.  As such, they run a high
43       risk of causing inconsistency between disk and memory data.  In support
44       of synchronization, a rollback calls each file managers ShutdownUsers
45       function to cause all of the applicaiton attached to the file manager
46       to unload its data.  After the rollback is complete, the file managers
47       RestartUsers virtual function is called to trigger the applications
48       to reload memory data from disk.
49
50       Synchronization services are not strictly required, but if they are commit
ed,
51       the application must take special care to insure that commit and rollback
52       are only used when in a stable state.
53
54
55   Event Types:
56
57     SavePoint - Savepoints are special events which indicate that this
58       point in the application may be desired at a later point.  Each
_squibm_____C:\TMP\VT\RBJRN.CPP_____Mon Sep 30 21:33:18 1996_
59       time the file is completely committed an initial savepoint identified
60       by savepoint key of 0 is written to the file.  When the file is
61       committed to savepoint 0, the event journal is erased completely and
62       file managers may be added or deleted.  If outstanding transactions
63       exist, then an error is thrown if an application attempts to modify
64       the set of file managers there are uncommitted transactions.
65
66     Overtype - An overtype event occurs when a place in a current file is
67       overtyped.  The information which was overtyped is lifted from the
68       original file and is saved in the rollback journal.  If a rollback
69       is requested, the overtypes are reversed by using the saved text
70       to replace the updated text in the base file.
71
72     Extend - An extend occurs when a base file is made longer.  The file
73       managers maintain a variable containing their original length.  When
74       a write event causes the base file to be extended beyond its original
75       length it is not necessary to retain any additional information.  The
76       base file can be reverted to its original state by truncating the
77       information which was appended.  As such an extend event only contains
78       the original file length.  Reverting from an extend causes the base
79       file to be truncated.
80
81     Header - The event log contains a series of header records.  The header
82       records contain the filenames of the files which the event journal
83       supports.  The header records are only used in the event of a program
84       abort.  If the program aborts, the names contained in the headers
85       enable the rollback manager to create temporary file managers to
86       revert the physical files when the rollback manager is started during
87       the next program run.
88
89     This object is designed to record events from either mode.  The primary
90     mode is for reversion the primary need at the time of this writing.
91
92     The Strategy of this class is to create and manage an event journal
93     formatted as follows:
94
95     |_EventDescriptor_|                  // Sizeof(event)
96     |_EventData_|                        // EventDescriptor.Size
97     |_EventTrailer_|                     // sizeof( EventDescriptor.Size )
98
99     |_EventDescriptor_|                  // Sizeof(event)
100    |_EventData_|                        // EventDescriptor.Size
101    |_EventTrailer_|                     // sizeof( EventDescriptor.Size )
102
103    The event trailers contain the total number of bytes contained in
104    the event for easier reverse navigation
105
106    An Event, described by an EventDescriptor describes a change to a base
107    file.  In reversion, the recorded event undoes the change (Must be played
108    backwards).  In replay, the recorded repeats the change to the subject
109    file (Must be played forward).
110
111    Warning:
112      All objects involved in a transaction must be created and attached
113      before you issue a write to any one of them.  You cannot attach
114      an object to the event journal if it is uncommitted.
115
116  Notes:
117    Events are captured whenever a file manager is attached to the event
118    journal.  This can cause some difficulties at shutdown.  It is normal
```

3

```
for an application to issue a flush during shutdown.  The flush may occur
after a commit, creating events in the rollback journal [illegible]
backed out after startup.  There are two solutions to this problem.

  1)  Write the shutdown routine so that it does not issue any writes.
      Since the commit function calls FlushAllUsers, all of the data relative
      to an object should have been recorded, and a flush isn't necessary

  2)  The second solution is to disconnect the event journal after the
      file is committed, effectively disregarding the writes which occur
      during the shutdown process.

Synchronization Concerns:
  There are two approaches to using the file_mgr/RollbackJournal class to
  control the state of target files.  Either design an application so
  that commit and rollback only occur at expected times, or provide
  virtual methods to the file_mgr which synchronize the disk and memory,
  prior to commit.

Since the event manager rollback services are only aware of the file manager
and have no means to determine whether the objects driving the file manager
have cached data, or whether the object driving the file manager is in a
reloadable state, measures must be taken to insure consistency of memory and
file contents at key points.

The key points are:
  At Commit - At commit time, it is presumed that the objects using a file
    are in a consistent state.  That is if all of the data which the objects
    contains were reloaded from disk, the object would come back in a usable
    state.  In database terms, this requirement means that commits must only
    occur when a record is in a consistent state (has been completed or
    logically aborted).  If the database instance crashed just after a commit
    where only part of a record were added, the database would not be able
    to handle the partial record.  In support of this requirement, the
    ObjectNotRestorableFlag in File_Mgr causes an error to be thrown when
    a commit is attempted on a file which would restore the object set
    to an inconsistent state.  It remains the applications responsibility
    to set this flag.

  Before a rollback - Because a rollback may make changes to the physical
    file, and an application may have objects instances which have informatio
n
    in memory which may have been dependent on changed sections of the file's

    data, there is a good chance that a rollback will cause the disk data and
    in-memory data to become inconsistent.  As such, there are virtual method
s
    in the file_mgr which effectively shut down any objects which depend on
    the physical file before the rollback is executed.

After a rollback -  After a rollback completes, the object set which depend
s
    on the physical file may be out of date.  As such there is a virtual meth
od
    in the file method which enables the objects to be reloaded from disk aft
er
    a rollback completes.

The two working strategies are to support the ObjectRestorable flag in the fil
e
squibm
```

4

```
  manager or to ensure that the application only calls commit and rollback
  [illegible]application is in a stable state.

  Savepoints 1/18/95
    Savepoints are events which get logged which enable the rollback operation
    to stop when a particular savepoint is achieved.  This enables part
    of a transaction to reversed without going back to the last committed
    state.
*/

#include <mgr_file.h>
#include <auxsvc.h>
#include <sys\stat.h>
#include <strstrea.h>
#include <stdlib.h>

// #include  <fstream.h>

////////////////////////////////////////////////////////////////////////
/////////   Member Functions for:   Event Journal    (event.cpp)  //////////
////////////////////////////////////////////////////////////////////////

//RollbackJournal::RollbackJournal() : FlexList( sizeof( File_Mgr* ) ),

RollbackJournal::RollbackJournal() : CL<File_Mgr>(  ),
  EventDataBuffer( MAX_EVENT_SIZE )
{
  RFD            = 0;       // File descriptor for Rollback file
  here           = 0;
  NextHandle     = 1;
  ActiveSavePoint = 0;
  CommittedFlag = 1;
  //TSB            = NULL;
}


RollbackJournal::~RollbackJournal(void)
{
  // HeaderSize = 0;
  Close();  // Close Rollback File
}

/*
  Create savepoint writes a savepoint event into the event log
  and increments the active savepoint counter.  The user application
  must retain the savepoint if rollback to that point is required
*/
long RollbackJournal::CreateSavePoint() {

  // All of the attached applications must flush so the data at the
  // savepoint is valid
/* for ( int i = 1; i <= (int)Nodes(); i++ ) {
    File_Mgr* FM;
    recalID( &FM, i );
    if ( FM ) {
      // Check if the objects attached to the file manager are marked restorable
      if ( FM->isObjectRestorable() == 0 ) {
        ProgramError PE( "RollbackJournal::Commit",
                    "Object is not in restorable state... Cannot commit" )
```


NOV 14 '97 16:52 FR BAKER & MCKENZIE 212 759 9133 TO 15561045011#9160 P.111

```
231         throw PE;
232       }
233       // Flush all objects which use the file manager
236       FM->FlushAllUsers();
235       FM->ResetExtended();
236     }
237   }
238 */
239   CL_forEach( *this ) {
240     File_Mgr* FM = get();
241           // Check if the objects attached to the file manager are marked restor
    able
242     if ( FM->IsObjectRestorable() == 0 ) {
243       ProgramError PE( "RollbackJournal::Commit",
244                        "Object is not in restorable state... Cannot commit" );
245       throw PE;
246     }
247     // Flush all objects which use the file manager
248     FM->FlushAllUsers();
249     FM->ResetExtended();
250   }
251   StartEvent();
252   Event.Type     = LIBFILE_EVENT_SAVEPOINT;
253   Event.Position = ++ActiveSavePoint;
254   EndEvent();
255   //if ( TSB )
256   //  TSB->OnCreateSavePoint( ActiveSavePoint );
257
258   return ActiveSavePoint;
259
260
261 }
262 /*
263   The earlier incarnation of this logic was reversion file specific.
264   The reverse traversal process was jeperdized by a partial write
265   event because the partial write would/could leave an erroneous trailer
266   at eof.
267
268   This collection of functions holds the content of the event in memory
269   until the event is complete.  Also a header and trailer are applied
270   to each event enabling forward or reverse event traversal.  In extreme
271   circumstances, where the reverse event journal ends with an erroneous
272   element, the event set can be forward traversed and the erroneous element
273   discarded.
274 */
275 //
276 //  StartEvent initializes the internal event descriptor buffer
277 //
278 //  Possible Errors:
279 //    None
280 //
281 //  The first event to a journal causes the currently attached
282 //  files managers to be flushed to a header.  After this header
283 //  is recorded, it is not permitted to change the connections
284 //  until the next Commit.
285
286 void RollbackJournal::StartEvent(){
287   // Initialize the header if one is missing
288   if ( CommittedFlag == 1 ) {
289     // Record the event journals header
```



```
290     FlushHeader();
291     // And Object is no longer committed
292     CommittedFlag = 0;
293   }
294
295   // If there is event data stored null it out
296   if ( EventDataSize ) {
297     memset( EventDataBuffer.Ptr(), 0, EventDataSize );
298     EventDataSize = 0;
299   }
300
301   Event.Clear();          // Clear the Event descriptor
302 }
303
304 //  SetEventData
305 //  This function transfers event data into the event Data buffer so
306 //   that the EndEvent can record it
307 //  Possible Errors
308 //    1) Cnt is larger than the EventBufferSize (Programming error)
309
310 void RollbackJournal::SetEventData( void* EventData, int Cnt ){
311   if ( Cnt > MAX_EVENT_SIZE )
312     throw( ProgramError( "RollbackJournal::SetEventData",
313                          "Data Block is larger than transfer buffer" )
314     );
315   memcpy( EventDataBuffer.Ptr(), EventData, Cnt );
316   EventDataSize = Cnt;
317 };
318
319 //
320 //  SetEventDescriptor
321 //  Event Data can be transferred to the RollbackJournal in any order.
322 //  An event descriptor may describe an event without a data component
323 //    as such there may be no EventData
324 void RollbackJournal::SetEventDescriptor( EventDescriptor& ED ){
325   Event = ED;
326 }
327 //
328 //  EndEvent
329 //  1)  Transfer the event header to the file
330 //  2)  Write the event data to the file
331 //  3)  Write the event trailer to the file
332 //
333 //  Possible Errors:
334 //  1)  Write error recording event
335 //
336 void RollbackJournal::EndEvent() {
337   long EventStart = Seek( 0, SEEK_END );
338   // Record the event header
339   try {
340     // How large is the entire event including header and trailer
341     Event.Size = sizeof( Event ) + EventDataSize + sizeof( Event.Size );
342     // Write out the event header
343     Write( &Event, sizeof( Event ) );
344     //  Write out the event data
345     Write( EventDataBuffer.Ptr(), EventDataSize );
346     //  Pad the event data to a longword
347     Write( "\0\0\0\0", PadSize( EventDataSize ) );
348
349     // Write out the event trailer
```

```
350     Write( &Event.Size, sizeof( Event.Size ) );
351
352 #if defined( _EVENT_JOURNAL_TRACE )
353     cerr << "Event Offset:  " << here - FullPadSize ( Event.Size ) << endl;
354     Event.printOn( cerr );
355     cerr.flush();
356 #endif
357   ) catch ( FileError ) {
358     // Truncate the file back to where the event started
359     Chsize( EventStart );
360     // Rethrow the exception
361     throw;
362   )
363 }
364 /*
365   Write the rollback managers header to disk:
366     -2) Confirm that there are no outstanding events
367     -1) Truncate the file
368     0) Reserve the first longword
369     1) Rewind the file manager list
370     2) Retrieve the file name name from each fmgr and write it to disk
371     3) Write each name with a space between
372     4) Revise the first longword to reflect the new header size
373
374   Possible Errors:
375     1) None ( Caught at lower levels )
376 */
377 void RollbackJournal::FlushHeader() {
378
379   Chsize( 0 );
380   Seek( 0, SEEK_SET );
381
382   // Clear the event record
383   Event.Clear();
384   // Set the header
385   Event.Type = LIBFILE_EVENT_HEADER;
386
387   //int CurNode = CurNum();
388   int CurNode = curNode();
389   CL_forEach( *this ) {
390     File_Mgr* FP = get();
391     if ( strlen( FP->GetName() ) == 0 ) continue;
392
393     // Set up the event parameters
394     Event.Handle  = FP->Handle;
395     EventDataBuffer.clear();
396
397     // Transfer the events data into the data buffer
398     SetEventData( FP->GetName(), strlen( FP->GetName() ) + 1 );
399
400     // Write the current event to the file
401     EndEvent();
402   }
403   // Write an initial savepoint marker
404   ActiveSavePoint = 0;
405   Event.Clear();
406   SetEventData( "", 0 );
407   Event.Type      = LIBFILE_EVENT_SAVEPOINT;
408   Event.Position  = ActiveSavePoint;
409   EndEvent();
```



```
410
411   // Go back to the original node
412   SetCurNode( CurNode );
413 }
414
415
416
417 ///AttachFmgr/////////////////////////////////////////////////////////////
418 //
419 // Attaches a File Manager to an RollbackJournal FlexList node.
420 // NOTE:  The File Manager's Handle MUST be set before calling
421 // this function.
422 //
423 // Possible Errors:
424 //  Fmgr is null
425 //  insD fails to allocate ( caught by new_handler )
426
427 void RollbackJournal::AttachFmgr ( File_Mgr *Fmgr )
428 {
429   // Fmgr->AttachJournal( this );
430   // Advance to the end of the list
431   //mkcur( Nodes() );
432
433   // Add this pointer to the list
434   //insD( &Fmgr );
435   insD( Fmgr );
436
437   // Set the file managers handle to the node value
438   Fmgr->SetHandle( GetNextHandle() );
439
440   // Write the headers out to disk
441   //  FlushHeader();
442 }
443
444 ///Chsize/////////////////////////////////////////////////////////////////
445 //
446 // Truncates the Rollback file to a specified length
447 //
448 // Possible Errors:
449 //   RFD is unset (Program usage error)
450 //   SZ is wacko
451 //   Seek error
452 void RollbackJournal::Chsize( long SZ)
453 {
454   if ( RFD <= 0 )
455     throw( ProgramError( "RollbackJournal::Chsize",
456                          "Invalid file Descriptor" ) );
457
458   if ( chsize( RFD, SZ ) ) {
459     FileError FE( "RollbackJournal::Chsize" );
460     FE.SetFileName( RbName );
461     FE.SetErrCode( FILE_CLOSE_ERROR );
462     throw( FE );
463   }
464
465   Seek( 0, SEEK_END );
466 }
467
468
469 ///Commit/////////////////////////////////////////////////////////////////
```

```
470 //
471 // Deletes all transactions contained in the Rollback file.
472 // Rollback file should be setup to handle new transactions.
473 //
474 // Possible Errors:
475 //  1)  File not opened first (programmer error)
476 void RollbackJournal::Commit()
477 {
478   if ( RFD <= 0 )
479     throw(
480       ProgramError( "RollbackJournal ->Invalid File Descriptor during commit" )
481     );
482
483
484   // Flush the file headers
485   // FlushHeader();
486
487   // Reset each of the File Mgr's 'Extended' flag to OFF?
488   CL_forEach( *this ) {
489   // for ( int i = 1; i <= (int)Nodes(); i++ ) {
490     File_Mgr* FM = get();
491     // recallID( &FM, 1 );
492     if ( FM ) {
493       // Check if the objects attached to the file manager are marked restorable
494       if ( FM->IsObjectRestorable() == 0 ) {
495         ProgramError PE( "RollbackJournal::Commit",
496                          "Object is not in restorable state... Cannot commit" )
    ;
497         throw PE;
498       }
499       // Flush all objects which use the file manager
500       FM->FlushAllUsers();
501
502       // Reset the extended flag
503       FM->ResetExtended( );
504     }
505   }
506
507   // Truncate the file
508   Chsize( 0 );
509   // Reset the savepoint
510   ActiveSavePoint = 0;
511   // Mark the journal committed
512   CommittedFlag = 1;
513
514   //if ( TSB )
515   //  TSB->OnCommit( );
516 }
517 // ShutDown ////////////////////////////////////////////////////////
518 //
519 //  When finished with a successful session, there is no need to
520 //  maintain any content to the event journal.
521 //
522 //  Strategy:
523 //    1)  Close the file
524 //    2)  Unlink it
525 void RollbackJournal::ShutDown() {
526   Close( );
527   unlink( RbName );
528 }
```



```
529
530 //Close////////////////////////////////////////////////////////
531 //
532 // Closes Rollback File
533 //
534 // Possible Errors:
535 //   1) Close fails
536 //   2) File managers have not been shut down
537 void RollbackJournal::Close(void) {
538   if ( nodes() ) {
539     throw ProgramError( "RollbackJournal::Close",
540                         "Close File_Mgrs before closing RollbackJournal" );
541   }
542
543   if ( RFD != -1 ) {
544     // cout << "Closing Event Journal" << endl;
545     if ( close( RFD ) ) {
546       FileError FE( "RollbackJournal::Close" );
547       FE.SetFileName( RbName );
548       FE.SetErrCode( FILE_CLOSE_ERROR );
549       throw( FE );
550     }
551     RFD = -1;
552   }
553 }
554
555
556 ///ConnectFmgr//////////////////////////////////////////////////////
557 //
558 // Connects (makes current) the specified File Manager which is
559 // contained in the class's FlexList.
560 //
561 // Possible Errors:
562 //  1)  Selection references an out of range handle
563 //  2)  Selected file manager has an invalid file descriptor
564 //
565 // Returns:
566 //  File descriptor for selected file manager or zero if
567 //  the file manager is a null placeholder
568
569 int RollbackJournal::SelectFmgr( int Hdl ) {
570   CL_forEach( *this ) {
571     if ( get()->Handle == Hdl )
572       return get()->GetFD();
573   }
574   return 0;
575   //File_Mgr* FP;
576   //mkcur();
577   //while ( next0( &FP ) ) {
578   //  if ( FP->Handle == Hdl ) return FP->GetFD();
579   //}
580   //return 0;
581 }
582
583
584 //Create///////////////////////////////////////////////////////////
585 //
586 // Creates the Rollback file and sets it's file descriptor: RFD
587 //
588 // Possible Errors:
```

```
589 //   1)  File error on create
590 //
591 void RollbackJournal::Create( )
592 {
593   // char* CurDir = getcwd( NULL, 0 );
594   // unlink( RbName.Ptr() );
595   RFD = open(RbName.Ptr(),
596           O_CREAT | O_TRUNC | O_RDWR | O_BINARY,
597           S_IREAD | S_IWRITE );
598   if( RFD <= 0 ) {
599     FileError FE( "RollbackJournal::Create", "File Creation Failed" );
600     FE.SetFileName( RbName );
601     FE.SetErrCode( FILE_CREATE_ERROR );
602     throw( FE );
603   }
604   // Many file systems get confused about binary mode
605   // Make sure we got opened correctly
606 #if defined ( __BORLANDC__ ) || defined( WIN32 )
607   write( RFD, "\n", 1 );
608   // Win95 is wacked.  Stat doesn't return the correct size!
609   // Try the old dos trick to get the stat info up to date
610   int DupRfd = dup( RFD );
611   close( DupRfd );
612
613   struct stat sb;
614   fstat( RFD, &sb );
615   if ( sb.st_size != 1 ) {
616     throw ProgramError( "RollbackJournal::Create", "File opened in text mode" );
617   }
618   chsize( RFD, 0 );
619 #endif
620
621 }
622
623
624 ///DestroyFmgrs////////////////////////////////////////////////////////////
625 //
626 // Destroys the TEMPORARY File Managers created in StartRollback().
627 //
628 // Notes:
629 //   the FM->Close() method removes the file manager reference from
630 //   the event journal class.  therefore rewinding the list each time
631 //   and calling next causes all of file managers to be deleted
632 //
633 // Possible Errors
634 //  0) None... FM->Close()
635 void RollbackJournal::DestroyTmpFmgrs ( )
636 {
637   // We have to use a static iterator because
638   // FileMgr->close may cause items dropped from
639   // the container
640   File_Mgr* FM;
641   while ( FM = top() ) {
642     FM->Close();
643   }
644 }
645
646 ///Open////////////////////////////////////////////////////////////////////
647 //
648 // Opens the Rollback File for read/write operations
```



```
649 //
650 // Possible Errors:
651 //   1)  Open fail
652 void RollbackJournal::Open( )
653 {
654   RFD = open((const char *)RbName, O_RDWR | O_BINARY,
655           S_IREAD | S_IWRITE );
656
657   if( RFD <= 0 ) {
658     FileError FE("RollbackJournal::Open", RbName);
659     FE.SetErrCode( FILE_OPEN_ERROR );
660     throw FE;
661   }
662   Seek ( 0, SEEK_SET );
663 }
664
665
666 ///Read////////////////////////////////////////////////////////////////////
667 //
668 // Read buffer from Rollback File
669 //
670 // Possible Errors:
671 //   1) Device error on read
672 //
673 // Returns:
674 //    Number of characters read
675 int RollbackJournal::Read( void *buf, int Get )
676 {
677   int Got = ::read( RFD, (char*)buf, Get );
678   if (Got == -1 ) {
679     FileError FE( "RollbackJournal::Read", RbName );
680     FE.SetErrCode( FILE_READ_ERROR );
681     throw FE;
682   }
683   return Got;
684 }
685
686 ///RollbackExist///////////////////////////////////////////////////////////
687 //
688 // Check to see if an Rollback File exists
689 //
690 // Possible Errors:
691 //    none
692 int RollbackJournal::RollbackExist()
693 {
694   if ( access( RbName, 0 ) == 0 ) {
695     struct stat sb;
696     stat( RbName.Ptr(), &sb );
697     if ( sb.st_size > 0 ) { return YES; }
698   }
699   return NO;
700 }
701
702
703 ///Seek////////////////////////////////////////////////////////////////////
704 //
705 // Seeks to a physical position within the Rollback File
706 //
707 // Possible Errors:
708 //    1)  File Error On Seek
```

```
709 //
710 // Return Value:
711 //   Current position relative to begin of file
712
713 long RollbackJournal::Seek( long where, int whence )
714 {
715   // Check to see if we're writing anyplace other than 'here'
716   int seekflag = ON;
717   switch ( whence ) {
718     case SEEK_SET:
719       if ( here == where ) seekflag = OFF;
720       break;
721     case SEEK_CUR:
722       if ( 0 == where ) seekflag = OFF;
723       break;
724   }
725   if ( seekflag ){
726     where = lseek( RFD, where, whence );
727     if ( where == -1 ) {
728       FileError FE( "RollbackJournal::Seek", RbName );
729       FE.SetErrCode( FILE_SEEK_ERROR );
730       throw FE;
731     }
732     here = where;
733   }
734   return(here);
735 }
736
737 ///SetName//////////////////////////////////////////////////////////////////
738 //
739 // Set the Rollback file's name
740 //
741 void RollbackJournal::SetName( char* DatabaseName )
742 {
743   RbName = DatabaseName;
744   // RbName += ".r";
745 }
746
747
748 /*  ReadHeaders
749     This function sets up for the Rollback process after an instance has
750     crashed.  Normal operation leaves the reversion stack empty.  When
751     an instance starts and finds an intact reversion stack, it is presumed
752     that the prior instance crashed and all of the reversion info must
753     be backed out.
754
755   1)  Go to the beginning of the file
756   2)  Read the size of the header
757   3)  Load the entire header block into a buffer
758   4)  Create a file manager for each name
759   5)  Set the name for each manager
760   Arguments:
761     IgnoreMissingFlag - A subject file may have disappeard
762       since the wipeout.  IgnoreMissing enables the balance
763       of the tables to be recovered by marking the object
764       pointer NULL in the container set.
765       (Rollback events skip objects for null pointers)
766   Possible Errors:
767     0) Failure to read bytes for headersize
768        (not an error... empty file is committed state
```



```
769     1) Failure to allocate Header Space
770     2) FP Open failure
771 */
772 void RollbackJournal::ReadHeader( int IgnoreMissingFlag ) {
773
774   long NextEventPosition = 0;
775   do {
776
777     NextEventPosition += LoadEvent( NextEventPosition );
778
779     if ( Event.Type != LIBFILE_EVENT_HEADER ) {
780       // Done reading names
781       break;
782     }
783
784     // No retrieval
785     if ( EventDataBuffer.length() == 0 ) break;
786
787     // Allocate the object and setup the name
788     File_Mgr* FP = new File_Mgr;
789     FP-> SetName( EventDataBuffer.Ptr() );
790     FP-> SetHandle( Event.Handle );
791
792     // Open the subject file
793     //  An error means that a file probably disappeared
794     try {
795       FP->Open();
796     } catch ( FileError& FE ) {
797       FE = FE;  // dummy assignment to stop compiler complaint
798       // Cancel the allocated object
799       //delete FP;
800       //FP = NULL;
801       if ( FP->GetFD() != -1 ) {
802         throw ProgramError( "RollbackJournal::ReadHeader",
803                             "Open fail should set FD to -1" );
804       }
805
806       // Missing is OKAY and the error was missing file
807       // if ( IgnoreMissingFlag == ON && errno == ENOENT ) {
808       if ( IgnoreMissingFlag == ON ) {
809         FP->EvnJrn = this;
810         ins0( FP );
811         errno = 0;
812         //ins0( &FP ); // Null content inserted
813         continue;
814       } else {
815         // If failure is not allowed, cleanup all allocated
816         // file objects and abort
817         CL_forEach( *this ) {
818           delete get();
819         }
820         oldNew();  // Clear the container
821         /*
822         mkcur();
823         // Clean Up Allocated objects
824         while( next0( &FP ) ){
825           delete FP;
826           del0();
827         }
828         */
```

```
                // Rethrow the file error                                  15
                throw;
            }
        } // End of Catch ( FileError )
        // Attach the file Manager to the RollbackJournal
        inost FP );
        fP->EvnJrn = this;
    } while ( 1 );
}

///////////////////////////////////////////////////////////////////////////
//StartRollback
//
// Creates the Rollback (RB) file for opens an existing one).
//
// This routine is responsible for getting things going:
//   1) Revert any leftover transactions
//   2) Get the system ready to go for normal operations
//
// Arguments:
//   DbName - Used to set the name of the rollback file
//   SalvageFlag - Enables the Event Journal to ignore missing tables
//
// The backout process goes like this
//      1) Read the header
//      2) Create temporary file managers from the header's info
//      3) Go to the end of the reversion stack
//      4) Revert the file events as per the stack (Perform a Rollback)
//      5) Destroy temporary file Managers
//      6) Truncate Rollback file to 0
//
// Possible Errors:
//   0) None
//
void RollbackJournal::StartRollback (char * DbName, int SalvageFlag )
{
    SetName( DbName );  // Set the Rollback Files Name

    if ( RollbackExist() ) {   // If Rollback File exists
        // Open fail throws a file error upward
        Open ( );

        // Load the headers and fire up the temporary objects
        ReadHeader( SalvageFlag );

        // If ReadHeaders filled any nodes then revert
        if ( nodes() ) {
            Rollback();
            DestroyTmpMgrs();
        } else { // There may have been a header
            // Truncate the file
            Chsize( 0 );
        }
    } else {
        try { Create ( ); }
        catch ( FileError& FE ) {
            FE.Report();
            throw;
        }
    }
}

///////////////////////////////////////////////////////////////////////////
//Write
//
// Write buffer to the Rollback file
// Maintain the here variable
//
// Possible Errors:
//   0) Invalid file descriptor
//   1) System write fails full file system...
//
// Return:
//   Number of characters written
//
int RollbackJournal::Write( void *buf, int cnt )
{
    int Put = ::write( RFD, (char*)buf, cnt );
    if ( Put != cnt ) {
        FileError FE( "RollbackJournal::Write", DbName );
        throw FE;
    }

    here += Put;
    return Put;
}

///////////////////////////////////////////////////////////////////////////
//Rollback
//
// Reads the Rollback file in reverse order, applying changes
// to the appropriate location in the Table File
//
// Possible Errors:
//   1) Inappropriate file descriptor
//   2) Seek failure
//   3) Write failure
//   4) Rollback file corrupted
//
// Return Value
//

void RollbackJournal::Rollback ( long SavePoint )
{
    if ( RFD <= 0 )
        throw( ProgramError( "RollbackJournal::Rollback",
                             "Unopened rollback file" )
        );

    // Shut down any instances which are attached to the physical
    // file managers
    for ( int i = 1; i <= (int)Nodes(); i++ ) {
        FileMgr* FM = get(i);
        // Flush all objects which use the file manager
        if ( FM->GetFD() > 0 ) {
            FM->ShutDownUsers();
            // If the file is still active after shutdown, throw an error
            if ( FM->IsFileActive() ) {
                throw ProgramError( "Cannot Rollback while file is marked active",
                                    FM->GetName() );
```

```
949   }
950
951   // Find the length of the Rollback file
952   long EventLocation = Seek( 0, SEEK_END );
953
954   // check for transactions
955   // long BeginTransactions = HeaderSize;
956
957   while ( EventLocation > sizeof( Event.Size ) ) {
958
959     // Decrement the position to point at the event trailer
960     Seek( EventLocation - sizeof( Event.Size ), SEEK_SET );
961
962     // Clear the current event
963     Event.Clear();
964
965     // Read the Event Trailer
966     Read( &Event.Size, sizeof( long ) );
967
968     // Go back to the event descriptor position
969     EventLocation -= fullPadSize ( Event.Size );
970
971     LoadEvent( EventLocation );
972
973     // if Journal tracing is on
974   #if defined( EVENT_JOURNAL_TRACE )
975     cerr << "Rollback Event Offset: " << EventLocation << endl;
976     Event.printOn( cerr );
977     cerr.flush();
978   #endif
979
980     // if the Event.Type is a savepoint, check it against
981     // the target savepoint
982     if ( Event.Type == LIBFILE_EVENT_SAVEPOINT ) {
983       // We hit the target savepoint
984       if ( Event.Position == SavePoint ) {
985         // Reset the active savepoint to this value
986         ActiveSavePoint = SavePoint;
987
988         // If we found the target savepoint stop
989         break;
990       }
991       if ( Event.Position < SavePoint ) {
992         throw ProgramError( "RollbackJournal::Rollback"
993                             "Cannot revert to abandoned savepoint" );
994       }
995       // Ignore the savepoint
996       continue;
997     }
998
999   // FH may be zero if the table to which it referred
1000  // was unopenable.  Skip activity on files with 0 FD
1001  int FH = SelectMgr( (int)Event.Handle );
1002  if ( FH <= 0 )
1003    continue;
1004
1005  switch ( Event.Type ) {
1006    case LIBFILE_EVENT_EXTEND:
1007      // Truncate the base file
1008
_equibase
```

```
1009        ::chsize( FH, Event.Position );
1010        SetEndOfFileExtended();
1011        break;
1012      case LIBFILE_EVENT_OVERTYPE:
1013      {
1014        // Replace the overtyped data
1015        ::lseek( FH, Event.Position, SEEK_SET );
1016        int Put = ::write( FH, EventDataBuffer.Ptr(), EventDataSize );
1017        if ( Put != EventDataSize ) {
1018          File_Mgr* FP = get();
1019          //recallO( &FP );
1020          FileError FE( "RollbackJournal::Rollback", FP->GetName() );
1021          FE.SetErrCode( FILE_WRITE_ERROR );
1022          throw FE;
1023        }
1024        break;
1025      }
1026      default:
1027      {
1028        RollbackJournalError EJE( "Journal file corrupted" );
1029        EJE.SetErrCode( RollbackJournal_CORRUPT );
1030        throw EJE;
1031        break;
1032      }
1033    }
1034
1035  } // end while
1036
1037
1038  // If this was savepoint zero, mark the journal committed to
1039  // permit adjustment of attached files
1040  if ( SavePoint == 0 ) {
1041    CommittedFlag = 1;
1042    // Truncate the event file to empty
1043    Chsize( 0 );
1044  } else {
1045    // Truncate the events journal after the current event position.
1046    // EventLocation is now at the beginning of the current event
1047    EventLocation += Event.Size;
1048    // drop everything after the current event
1049    Chsize( EventLocation );
1050  }
1051
1052  // Reload any object instances which depend on the files content
1053  //for ( I = 1; I <= ((int)Nodes(); I++ ) {
1054  CL_ForEach( *this ) {
1055    File_Mgr* FM = get();
1056    // Force the file physically set to 0.  The above modifications
1057    // probably changed the file's offset from FM->Here
1058    if ( FM->GetFD() > 0 ) {
1059      FM->Seek( 0, SEEK_SET, OU );
1060      // Flush all objects which use the file manager
1061      FM->RestartUsers();
1062    }
1063  }
1064
1065  //if ( TSB )
1066  // TSB->UnHold(back( SavePoint );
1067
1068 }
```


000083

```
1069
1070 //
1071 //  Query committed returns yes if there are transactions
1072 //    in the stack
1073 //
1074 //  Possible Errors:
1075 //
1076 //  Returns 1 for committed
1077 //          0 for uncommitted
1078
1079 int RollbackJournal::CheckCommitted() {
1080   return CommittedFlag;
1081 }
1082
1083
1084 //
1085 // LoadEvent loads an event at an offset in the file
1086 //
1087 // Returns the total space consumed by the event
1088 //
1089 // Results:
1090 //   Loads the event structure with the current event
1091 //   Load the EventDataBuffer with the data of the current event
1092 //   Sets EventDataSize to the size of the data in the event
1093
1094 long RollbackJournal::LoadEvent( long EventPosition ) {
1095
1096   // Clear the event and event buffer
1097   Event.Clear();
1098
1099   Seek( EventPosition, SEEK_SET );
1100   int GotCnt = Read( &Event, sizeof( Event ) );
1101
1102   EventDataBuffer.clear();
1103   EventDataSize = Event.Size - sizeof( Event ) - sizeof( Event.Size );
1104
1105   // Load the event's data
1106   Read( EventDataBuffer.Ptr(), EventDataSize );
1107
1108   // Return the full size of the event
1109   return FullPadSize ( Event.Size );
1110 }
1111
1112 // untested
1113 void RollbackJournal::DropFileRgn( int Handle ) {
1114
1115   // Drop the node corresponding to Handle from the event manager
1116   CL_ForEach( *this ) {
1117     if ( get()->Handle == Handle ) {
1118       removeByPut();
1119       break;
1120     }
1121   }
1122 }
```

```
1 ////////////////////////////////////////////////////////////////////////
2 // treepath.h: Implements a stack of "footprints" thru a tree
3 // Copyright(c) 1995 Azarona Software. All rights reserved.
4 ////////////////////////////////////////////////////////////////////////
5 #include "treepath.h"
6
7 TreePath::TreePath(unsigned n)
8 // Sets up a stack with a maximum of n elements.
9 {
10   st = (Footprint *)new char[n*sizeof(Footprint)];
11   if (st && n) { // Array isn't null
12     fn = st+n;
13     dimen = n;
14   }
15   else { // Array is null, which is always "full"
16     fn = st;
17     dimen = 0;
18   }
19   top = fn;
20 }
21
22 TreePath::~TreePath()
23 {
24   Clear();
25   delete[] (char *)st;
26 }
27
28 void TreePath::Clear()
29 // Makes the stack go empty, destroying all used elements.
30 {
31   while (top != fn) {
32     top->Footprint::~Footprint();
33     top++;
34   }
35 }
36
37 int TreePath::Push(const Footprint &x)
38 // Puts a new element at the front of the stack, holding
39 // a copy of x. Returns 1 if successful, else 0 if no
40 // room in stack.
41 {
42   if (IsFull()) return 0;
43   top--;
44   new(top) Footprint(x);
45   return 1;
46 }
47
48 int TreePath::Pop()
49 // Removes the front element from the stack w/o copying
50 // the data there. If stack is empty, a 0 is returned,
51 // else a 1.
52 {
53   if (IsEmpty()) return 0;
54   top->Footprint::~Footprint();
55   top++;
56   return 1;
57 }
58
59 int TreePath::Rewind(Footprint *fp)
60 // Rewinds the stack until top equals the position
```

```
61 // represented by fp.
62 {
63   while (top != fp) {
64     top->Footprint::~Footprint();
65     top++;
66   }
67   return 1;
68 }
69
```

C:\TMP\VF\TREEPATH.CPP — Mon Sep 30 21:33:18 1996

000085

1

```
///////////////////////////////////////////////////////////////////
// treepath.h: Defines a stack of "footprints" thru a tree.
// Copyright(c) 1995 Azarona Software. All rights reserved.
///////////////////////////////////////////////////////////////////
#ifndef H_TREEPATH
#define H_TREEPATH
#include "coptrb.h"
#include "vcache.h"
#include "placenew.h"

// A footprint is a special type of Coptr, that holds not
// only the normal Coptrb stuff, (such as the location of
// a node), but also the position of an entry within the
// node. These footprints work in conjunction with the
// Vnode, VnodeBucket, VnodeCache, and TreePath classes.

class Footprint : public Coptrb {
// *** This FootPrint class based on Coptr<TYPE> template ***
public:
  int posn;
  // p should be defaulted here, but BC++ 3.1 doesn't like it
  friend void *operator new(size_t n, Footprint &cp, long p);
  void Delete();
  Footprint(VnodeCache &c, long p = 0);
  Footprint(const Footprint &c);
  void operator=(const Footprint &c);
  void operator=(long p);
  VnodeBucket &operator*();
  VnodeBucket *operator->();
};

// p should not be defaulted here, but that's what BC++ 3.1 likes

inline void *operator new(size_t, Footprint &cp, long p=0)
// Allocates a new object to reside at address p in the file.
// If p is non-zero, the object is presumed to already
// allocated. The allocation takes place by calling Alloc(),
// which not only allocates the object on disk, but also
// reserves a cache bucket for it. A Bucketb pointer to this
// bucket is returned by Alloc(), which we typecast as a
// VnodeBucket pointer. Then, we do a clever typecast
// to a TYPE pointer, which gives us a pointer to the bucket's
// data. This pointer is returned, and is passed on to the assoc-
// iated TYPE constructor as the 'this' pointer. (We had to pass
// the pointer back as a void pointer to satisfy the syntax for
// the new operator.)
// NOTE: We don't use the size_t parameter to determine how
// many bytes to allocate. Instead, we use the data_size stored
// in the cache.
{
  return (Vnode *)((VnodeBucket *)
         (cp.Alloc(((VnodeCache *)(cp.cache))->NodeSize(), p)));
}

inline void Footprint::Delete()
// Frees the data associated with the Coptr c. The Coptr is
// then set to null.
{
  Coptrb::Delete(((VnodeCache *)cache)->NodeSize());
}
```


2

```

inline Footprint::Footprint(VnodeCache &c, long p)
// General Footprint constructor. Note that a cache
// is required. Like all coptr's, the footprint stays
// unbound till needed.
: Coptrb(c, p)
{
  posn = BEFORE_ENTRIES;
}

inline Footprint::Footprint(const Footprint &c)
// Footprint copy constructor. Like all coptr's, the destination
// footprint won't be bound until needed, even if source is bound.
: Coptrb(c)
{
  posn = c.posn;
}

inline void Footprint::operator=(const Footprint &c)
// Footprint assignment
{
  Copy(c);
  posn = c.posn;
}

inline void Footprint::operator=(long p)
// Another footprint assignment, where source is the
// new file address. Like all coptr's the resulting
// footprint stays unbound until needed.
{
  Release();
  addr = p;
  ///// posn = BEFORE_ENTIRES; DONT DO <<< WHY NOT? I FORGOT REASON >>>
}

inline VnodeBucket &Footprint::operator*()
// Access to the referenced bucket. If the pointer
// isn't bound to a bucket one is grabbed for it.
{
  if (!bound) Bind();
  return *(VnodeBucket *)bkt;
}

inline VnodeBucket *Footprint::operator->()
// Access to the referenced bucket. If the pointer
// isn't bound to a bucket, one is grabbed for it.
{
  if (!bound) Bind();
  return (VnodeBucket *)bkt;
}

///////////////////////////////////////////////////////////////////

class TreePath {   // A stack of footprints
protected:
  Footprint *st;     // Pointer to start element
  Footprint *fn;     // Pointer to one past last element
  Footprint *top;    // Pointer to top of stack
  unsigned dimlen;   // Maximum number of elements
public:
```

```
121   TreePath(unsigned n);
122   virtual ~TreePath();
123   void Clear();
124   void Reset(const Footprint &s);
125   int Push(const Footprint &s);
126   int Pop();
127   int Rewind(Footprint *fp);
128   Footprint *Top();
129   Footprint &Curr();
130   Footprint &Parent();
131   int IsEmpty() const;
132   int IsFull() const;
133   unsigned Size() const;
134 };
135
136
137 inline int TreePath::IsEmpty() const
138 {
139   return top == fn;
140 }
141
142 inline int TreePath::IsFull() const
143 {
144   return top == st;
145 }
146
147 inline Footprint *TreePath::Top()
148 {
149   return IsEmpty() ? 0 : top;
150 }
151
152 inline Footprint &TreePath::Curr()
153 // for speed: ASSUMES stack isn't empty.
154 {
155   return *top;
156 }
157
158 inline Footprint &TreePath::Parent()
159 // for speed: ASSUMES path isn't empty, and that
160 // current node really does have a parent.
161 {
162   return *(top+1);
163 }
164
165 inline unsigned TreePath::Size() const
166 {
167   return dfnlen;
168 }
169
170 inline void TreePath::Reset(const Footprint &s)
171 {
172   Clear();
173   Push(s);
174 }
175
176 #endif
177
178
179
180
```

```
1  ////////////////////////////////////////////////////////////////////////////
2  // vcache.cpp: Cache for vnode bucket implementation.
3  // Copyright(c) 1993 Asarona Software. All rights reserved.
4  ////////////////////////////////////////////////////////////////////////////
5  #include "vcache.h"
6  #include "placenew.h"
7
8  ////////////////////////////////////////
9  // Vnode bucket methods
10 ////////////////////////////////////////
11
12 void VnodeBucket::Fetch(Fmgr &f)
13 // Fetch object data from the file f
14 {
15    f.Fetch((Vnode *)this, data_size, addr);
16    dirty = 0;
17 }
18
19 void VnodeBucket::Store(Fmgr &f)
20 // Stores object data into the file f
21 {
22    f.Store((Vnode *)this, data_size, addr);
23    dirty = 0;
24 }
25
26 int VnodeBucket::RoomFor(const EntryD &e)
27 // Returns 1 if there's room in the node for entry e.
28 {
29    return  tail + e.Size() <= (unsigned)NodeSpace();
30 }
31
32 int VnodeBucket::RoomFor(int n)
33 // Returns 1 if this node has room for an entry of size n.
34 {
35    return tail + n <= NodeSpace();
36 }
37
38 int VnodeBucket::IsHungry()
39 // Returns 1 if this node wants more entries
40 {
41    return tail < NodeSpace() / 2;
42 }
43
44 int VnodeBucket::IsSatisfied()
45 // Returns 1 if this node has enough entries
46 {
47    return tail > NodeSpace() / 2;
48 }
49
50 int VnodeBucket::IsFat()
51 // Returns 1 if this node has more than enough entries
52 {
53    return PrevPosn(tail) > NodeSpace() / 2;
54 }
55
56
57 ////////////////////////////////////////
58 // Vnode cache methods
59 ////////////////////////////////////////
60
```

```
61
62 // Constructing (VnodeCache::ConstructBuckets(int nb, int ns)
63 // Allocate room for nb buckets with a node size of ns,
64 // and then construct them in-place. Note that CacheB will
65 // take care of a lot of the initialization later.
66 {
67    int bkt_size = sizeof(VnodeBucket) + ns - 1;
68
69    node_size = ns;
70
71    char *p = new char[nb*bkt_size];
72
73    char *q = p;
74
75    for (int i = 0; i<nb; i++) {
76       new(q) VnodeBucket(ns);
77       q += bkt_size;
78    }
79
80    return buckets = (VnodeBucket *)p;
81 }
82
83 VnodeCache::VnodeCache(int nb, int ns)
84 // Set up a cache of nb Vnode buckets. Note that CacheB does
85 // a lot of the initialization for us.
86 : CacheB(ConstructBuckets(nb, ns), nb, sizeof(VnodeBucket)+ns-1)
87 {
88 }
89
90 VnodeCache::~VnodeCache()
91 {
92 //C700 from:  If (fptr) Flush();
93    if (fPtrId) Flush();
94    Disconnect();
95    // We must de-allocate buckets the way we allocated them, as
96    // just raw bytes. We ASSUME here that any bucket destructors
97    // either have already been called, or are not needed.
98    delete[] (char *)buckets;
99 }
100
101 void VnodeCache::Reconfigure(int ns)
102 // Reconfigures the cache to work with Vnodes of a new
103 // node size ns. The cache is made empty.
104 // WARNING: The cache better not have any references to
105 // it or an exception is thrown.
106 {
107    Clear(); // May throw dangling pointer exceptions
108    delete[] (char *)buckets;
109    ConstructBuckets(nbuckets, ns);
110    Setup(buckets, nbuckets, sizeof(VnodeBucket) + ns - 1);
111 }
```

```
1 ////////////////////////////////////////////////////////////////////////////
2 // vcache.h: Cache for vnode bucket class definition.
3 // Copyright(c) 1993 Azrona Software. All rights reserved.
4 ////////////////////////////////////////////////////////////////////////////
5 #ifndef H_VCACHE
6 #define H_VCACHE
7 #include "bucketb.h"
8 #include "cacheb.h"
9 #include "mgr.h"
10 #include "vnode.h"
11
12 class VnodeBucket : public BucketB, public Vnode {
13 // IMPORTANT! Vnode must be last base class, cause room for it
14 // is actually dynamically allocated at the end of the statically
15 // determined size. Also, no extra data members can be added here!
16 public:
17   VnodeBucket();
18   VnodeBucket(int ns);
19   int NodeSpace();
20   int RoomFor(const EntryB &e);
21   int RoomFor(int n);
22   int IsHungry();
23   int IsSatisfied();
24   int IsFat();
25   virtual void Fetch(Mgr &f);
26   virtual void Store(Mgr &f);
27 };
28
29 inline VnodeBucket::VnodeBucket()
30 // Calls default BucketB() and Vnode() constructors
31 {
32 }
33
34 inline VnodeBucket::VnodeBucket(int ns)
35 {
36   data_size = ns;
37 }
38
39 inline int VnodeBucket::NodeSpace()
40 // Returns how much space there is in the node for data
41 {
42   return data_size - sizeof(Vnode) + 1;
43 }
44
45 class VnodeCache : public CacheB {
46 protected:
47   VnodeBucket *buckets;
48   int node_size; // Size of node in each bucket
49   VnodeBucket *ConstructBuckets(int nb, int ns);
50 public:
51   VnodeCache(int nb, int ns);
52   virtual ~VnodeCache();
53   void Reconfigure(int ns);
54   int NodeSize();
55 };
56
57 inline int VnodeCache::NodeSize()
58 {
59   return node_size;
60 }
```

squiba C:\TMP\VT\VCACHE.H Thu Aug 10 03:45:46 1995

000089

```
 1 ////////////////////////////////////////////////////////////
 2 // Vnode.cpp: Variable-order btree node methods.
 3 // Copyright(c) 1992 Azarone Software, All rights reserved.
 4 ////////////////////////////////////////////////////////////
 5 #include <string.h>
 6 #include <fstream.h>
 7 #include <stdlib.h>
 8 #include <vtree.h>
 9
10 #ifndef min
11 #  define max(a,b)    (((a) > (b)) ? (a) : (b))
12 #  define min(a,b)    (((a) < (b)) ? (a) : (b))
13 #endif
14
15
16 /*
17    The Entryb structures KeyCode must be set to point to a
18    valid KeyCode vector.  Since each index has such a vector
19    the Entry must be redirected at the time of usage
20 */
21 void Entryb::SelectVtree( Vtree& VT ) {
22   VT.SetEntryKeyVector( *this );
23 }
24 ////////////////////////////////////////////////////////////
25 // Entryb methods
26 ////////////////////////////////////////////////////////////
27
28 //
29 //  Clear a node for the next operation
30 //
31 void Entryb::Clear(){
32   // Null out the data area
33   right = 0;            // Points to right child
34   //data_field = 0;       // Data field, usually a record number or offset
35   memset( Data, 0, ENTRYDATASPACE );
36   // Clear the key area
37   memset( key, 0, ENTRYKEYSPACE );
38   // How many actual bytes in this entry
39   EntrySize = BaseEntrySize();
40 }
41 //
42 // Set up the Data area
43 //
44 void Entryb::SetData( void* NewData, int Len ){
45   memcpy( Data, NewData, Len );
46 }
47 //
48 //  Print out the contents of a key
49 //
50 void Entryb::printOn( ostream& os ) {
51
52   // Print out the default key
53   if ( Entryb::KeyCode == NULL ) {
54     throw( "Entryb::printOn", "Key vector undefined" );
55   }
56
57   // Dump the key datafie
58   os << "s:" << Size() << "\t"
59      << "d:" << data_field << "\t"
60      << "r:" << right << "\t";
```



```
61
62
63   char* TargBuf = key;
64   int DataLen = 0;
65   while ( Entryb::KeyCode[i] ) {
66   switch ( Entryb::KeyCode[ i ] ) {
67     case VT_NULL:
68     break;
69     case INT32:
70       os << "L:" << *(__int32*) TargBuf << "\t";
71       TargBuf += sizeof( __int32 );
72       break;
73     case UINT32:
74       os << "UL:" << *(unsigned long*) TargBuf << "\t";
75       TargBuf += sizeof( unsigned long );
76       break;
77     case FLOAT_8:
78       os << "F:" << *(float*) TargBuf << "\t";
79       TargBuf += sizeof( float );
80       break;
81     case FLOAT_10:
82       os << "LD:" << *(long double *) TargBuf << "\t";
83       TargBuf += sizeof( long double );
84       break;
85     case INT64:
86       os  << hex
87         << "H64:" <<  *(__int32*) ( TargBuf + 4 )
88         <<  *(__int32*) ( TargBuf ) << "\t"
89         << dec;
90       TargBuf += sizeof( __int64 );
91       break;
92     case BINARY:
93     case CHAR:
94       DataLen = *(short*) TargBuf;
95       TargBuf += sizeof( short );
96       os << "C:";
97       os.write( TargBuf, DataLen );
98       os << "\t";
99       TargBuf += DataLen;
100      break;
101    default:
102      os << "Unknown Key type";
103    }
104    i++;
105  }
106  os << endl;
107 }
108 //
109 //  Move the key data to the Data area
110 //
111 void Entryb::GetKeyValue( int KeyIx, void* DataArea, short& Cnt ){
112
113   // Pointer to base data
114   // char* SrcBuf = key;
115   Cnt = 0;
116   char KeyType = 0;
117   char* SrcArea =  (char*)SeekElement( KeyIx, KeyType, Cnt );
118   if ( KeyType >= CHAR ) {
119     // If this is a variable length segment element, EntrySize the length
120     //  header and copy the data
```

```
        memmove( DataArea, SrcArea + sizeof( short ), Cnt );
    } else {
        memmove( DataArea, SrcArea, Cnt );
    }
}

//
//  Extend a composite key by adding the data
//

void Entryb::BuildKey( char KeyCode, void* KeyData, short DataLen ){

    // If we are starting with a blank key, initialize EntrySize
    if ( EntrySize == 0 ) EntrySize = BaseEntrySize();

   // Calculate the target area for the data
    char* TargBuf = key + EntrySize - BaseEntrySize();

   switch ( KeyCode ) {
    case VT_NULL:
      return;
    case INT32:
    case UINT32:
      DataLen = sizeof( __int32 );
      break;
        case INT64:
      DataLen = sizeof( __int64 );
      break;
    case FLOAT_8:
      DataLen = sizeof( float );
      break;
    case FLOAT_10:
      DataLen = sizeof( long double );
      break;
    case CHAR:
      DataLen = strlen( (char*)KeyData );
    case BINARY:
    default:
      memcpy ( TargBuf, &DataLen, sizeof( DataLen ));
      TargBuf += sizeof( DataLen );
      EntrySize    += sizeof( DataLen );
      break;
  }
  // Move the key data into the space
  memcpy ( TargBuf, KeyData, DataLen );
  EntrySize += DataLen;
  CheckOverrun();
  return;
}

//  AddKeyElement( long& Val )
//
//  This set of overloaded operators extend the current key
//  to include the referenced value.  If the application attempts
//  to add a type mismatching the keycode vector an error is thrown
//
//  Arguments:
//   1)  Reference to value to be added to key
//
//  Possible Errors:
```



```
//    1)  Type mismatches current slot
//
//  Strategy:
//    1)  Advance to the end key
//    2)  Check confirm the element type matches the function
//    3)  Add the key
//
void Entryb::AddKeyElement( long Value ) {
  char  ElementType = 0;
  short ElemSize = 0;
  ConfirmKeyCode();

  void* TargBuf = SeekElement( -1, ElementType, ElemSize );
  if ( ElementType != INT32 ) {
    throw ProgramError( "Entryb::BuildKey", "Type mismatch long" );
  }
  // Move the key data into the space
  memcpy ( TargBuf, &Value, sizeof( long ) );
  EntrySize += sizeof( long );
  CheckOverrun();
}

void Entryb::AddKeyElement( __int64 Value ) {
  char  ElementType = 0;
  short ElemSize = 0;
  ConfirmKeyCode();

  void* TargBuf = SeekElement( -1, ElementType, ElemSize );
  if ( ElementType != INT64 ) {
    throw ProgramError( "Entryb::BuildKey", "Type mismatch INT64" );
  }
  // Move the key data into the space
  memcpy ( TargBuf, &Value, sizeof( __int64 ) );
  EntrySize += sizeof( __int64 );
  CheckOverrun();
}

void Entryb::AddKeyElement( unsigned long Value ) {
  char  ElementType = 0;
  short ElemSize = 0;
  ConfirmKeyCode();

  void* TargBuf = SeekElement( -1, ElementType, ElemSize );
  if ( ElementType != UINT32 ) {
    throw ProgramError( "Entryb::BuildKey", "Type mismatch unsigned long" );
  }
  // Move the key data into the space
  memcpy ( TargBuf, &Value, sizeof( unsigned long ) );
  EntrySize += sizeof( unsigned long );
  CheckOverrun();
}

void Entryb::AddKeyElement( long double Value ) {
  char  ElementType = 0;
  short ElemSize = 0;
  ConfirmKeyCode();

  void* TargBuf = SeekElement( -1, ElementType, ElemSize );
  if ( ElementType != FLOAT_10 ) {
    throw ProgramError( "Entryb::BuildKey", "Type mismatch double long" );
```

```
241 }
242 // Move the key data into the space
243 memcpy ( TargBuf, &Value, sizeof( long double ) );
244 EntrySize += sizeof( long double );
245 CheckOverrun();
246 }
247
248 void Entryb::AddKeyElement( void* Ptr, short& DataLen ) {
249   char ElementType = 0;
250   short ElemSize = 0;
251   ConfirmKeyCode();
252   void* TargBuf = SeekElement( -1, ElementType, ElemSize );
253   if ( ElementType != BINARY ) {
254     throw ProgramError( "Entryb::BuildKey", "Type mismatch void" );
255   }
256   memcpy ( TargBuf, &DataLen, sizeof( DataLen ) );
257   TargBuf = (char*)TargBuf + sizeof( DataLen );
258   EntrySize   += sizeof( DataLen );
259
260   // Move the key data into the space
261   memcpy ( TargBuf, Ptr, DataLen );
262   EntrySize += DataLen;
263   CheckOverrun();
264 }
265
266 void Entryb::AddKeyElement( char* Ptr ) {
267   char ElementType = 0;
268   short ElemSize = 0;
269   ConfirmKeyCode();
270
271   void* TargBuf = SeekElement( -1, ElementType, ElemSize );
272   if ( ElementType != CHAR ) {
273     throw ProgramError( "Entryb::BuildKey", "Type mismatch char" );
274   }
275   short DataLen = strlen( Ptr );
276   memcpy ( TargBuf, &DataLen, sizeof( DataLen ) );
277   TargBuf = (char*)TargBuf + sizeof( DataLen );
278   EntrySize   += sizeof( DataLen );
279
280   // Move the key data into the space
281   memcpy ( TargBuf, Ptr, DataLen );
282   EntrySize += DataLen;
283   CheckOverrun();
284 }
285
286 void Entryb::AddKeyElement( unsigned char* Ptr ) {
287   char ElementType = 0;
288   short ElemSize = 0;
289   ConfirmKeyCode();
290
291   void* TargBuf = SeekElement( -1, ElementType, ElemSize );
292   if ( ElementType != CHAR ) {
293     throw ProgramError( "Entryb::BuildKey", "Type mismatch uchar" );
294   }
295   short DataLen = strlen( (char*) Ptr );
296   memcpy ( TargBuf, &DataLen, sizeof( DataLen ) );
297   TargBuf = (char*)TargBuf + sizeof( DataLen );
298   EntrySize   += sizeof( DataLen );
299
300   // Move the key data into the space
301   memcpy ( TargBuf, Ptr, DataLen );
302   EntrySize += DataLen;
303   CheckOverrun();
304 }
305
306 // This function returns the index of the next blank element in the key
307 //
308 // Objectives:
309 //   Select a particular element
310 //   or Position to last element
311 //   Retrieve current data area
312 //   Retrieve current data type
313 //   Retrieve current element size
314 //
315 // Notes:
316 //
317 // Return Values:
318 //   Returns pointer to the target entry position
319 //   or NULL if the entry is full.
320 //
321 //   The ElementType receives a key type for the position in question
322 //   The DataLen receives the number of bytes which this key segment
323 //   requires.
324 //
325 //   If ElementNo is -1 then the key's content is set to the last unfilled
326 //   element position in preparation for extending the key by a value.
327 //
328 //   If the key is completely full, the ElementType is set to VT_NULL and a
329 //   null pointer is returned.
330 //
331 // Arguments:
332 //   ElementNo     - Index of element to seek to (-1 for last)
333 //   ElementType   - Reference variable returning the type of the element
334 //   DataLen       - Return of size of returned element
335 //
336 // Strategy:
337 //   1) If a cleared entry
338 //   2) Traverse elements to the end
339 //   3) Set
340 //
341 void* Entryb::SeekElement( int ElementNo,
342                            char& ElementType,
343                            short& DataLen,
344                            short* ElemSz )
345 {
346   if ( KeyCode == NULL ) {
347     throw ProgramError( "Entryb::SeekElement", "KeyCode not set for entry" );
348   }
349   // Byte index from beginning of key data
350   int CurPos = 0;
351   DataLen = 0;
352
353   // Index of element
354   int Element = 0;
355   // Traverse the key to the appropriate position
356   do {
357
358     // Are we beyond the valid data
359     if ( CurPos >= EntrySize - BaseEntrySize() ) {
360       // We have reached the end of the record
```


000092

```
361       break;
362     }
363
364     // Advance the read buffer
365     int AuxDataLen = 0;
366     switch ( KeyCode[ ElementIx ] ) {
367      case VT_NULL:
368        // entry is full
369        DataLen = 0;
370        ElementType = VT_NULL;
371        return NULL;
372      case INT32:
373      case UINT32:
374        DataLen = sizeof( __int32 );
375        break;
376      case INT64:
377        DataLen = sizeof( __int64 );
378        break;
379      case FLOAT_8:
380        DataLen = sizeof( float );
381        break;
382      case FLOAT_10:
383        DataLen = sizeof( long double );
384        break;
385      case CHAR:
386      case BINARY:
387 //     case WCCHAR:
388      default:
389        DataLen =  *(short*) (key + CurPos);
390        AuxDataLen = sizeof( short );
391        break;
392     } // switch ( KeyCode[ ElementIx ] )
393
394     if ( ElementNo == 0 ) break;
395
396     CurPos += DataLen + AuxDataLen;
397     ElementIx++;
398
399   }  while ( ElementNo-- );  // while ( CurPos < EntrySize )
400   ElementType = KeyCode[ ElementIx ];
401   char* TargBuf = key + CurPos;
402   if ( ElemIx ) *ElemIx = ElementIx;
403   return (void*)TargBuf;
404 }
405
406 /*
407   Return the number of elements defined in the key
408 */
409 short Entryb::GetElementCount() {
410   short ElemCnt = 0;
411   char ElemType;
412   short ElemSize;
413   SeekElement( -1, ElemType, ElemSize, &ElemCnt );
414   return ElemCnt;
415 }
416
417 /*
418   Duplicate the first number of segments of a key into
419   the current key
420 */
```



```
421 void Entryb::DupKey( Entryb& Ent, int DupCnt ) {
422
423   // Make the current key
424   Clear();
425   for ( int i = 0; i < DupCnt; i++ ) {
426     short SegSize;
427     char *SrcBuffer;
428     char ElemType;
429
430     SrcBuffer = (char*)Ent.SeekElement( i, ElemType, SegSize );
431     if ( ElemType >= Entryb::CHAR ) {
432       // Skip the stringsize header
433       SrcBuffer += sizeof( short );
434     }
435     BuildKey( ElemType, SrcBuffer, SegSize );
436   }
437 }
438
439 Entryb::Entryb()
440 // Set entry to "null", set EntrySize value to actual used size of
441 // entry, including null byte for key.
442 {  Clear();
443    KeyCode = NULL;
444 }
445
446 Entryb::Entryb( char* KeyVector )
447 {  Clear();
448    KeyCode = KeyVector;
449 }
450 Entryb::Entryb(const Entryb &src)
451 // Copy constructor copies data from source to this entry.
452 // ASSUMES there's enough room for all of src's data!
453 {
454   if ( src.Size() > ENTRYKEYSPACE ) {
455     throw ProgramError( "Entryb::Entryb(const Entryb &src)",
456                         "Overrun of key space" );
457   }
458   Clear();
459   KeyCode = src.KeyCode;
460   *this = src;
461   // memcpy(this, &src, src.Size());
462 }
463 Entryb::Entryb( Vtree& VT ) {
464   Clear();
465   VT.SetEntryKeyVector( *this );
466 }
467
468
469 void Entryb::operator=(const Entryb &src)
470 // Copies data from source to this entry.
471 // ASSUMES there's enough room for all of src's data!
472 {
473   if ( src.Size() > ENTRYKEYSPACE ) {
474     throw ProgramError( "Entryb::operator=", "Overrun of key space" );
475   }
476   right = src.right;
477   //data_field = src.data_field;
478   memcpy( Data, src.Data, ENTRYDATASPACE );
479   EntrySize  = src.EntrySize;
480   memcpy( key, src.key, src.Size() - src.BaseEntrySize( ) );
```

```
481   // memcpy((void*)this, &src, src.Size());
482 }
483
484 int CompVector ( void *a, void* b, int len ) {
485   unsigned char* A = (unsigned char*) a;
486   unsigned char* B = (unsigned char*) b;
487   while ( len && *B == *A ) { A++; B++; len--; }
488   if ( len == 0 ) return 0;
489   int x = *A - *B;
490   x = ( x > 0 ) ? 1 : x;  // 1 if greater
491   x = ( x < 0 ) ? -1 : x;
492   return x;
493 }
494
495 // Compare
496 //
497 // This function compares two entries.
498 //  Usually entries are completely defined.  (containing a valid item
499 //  for each KeyCode element)
500 //  However when searching for underdefined keys, a key element may only
501 //  contain leading elements.  When an entry is underdefined, compare
502 //  returns the partial result.
503 //
504 //  Returns:
505 //    0 if entries are equal
506 //    1 if entry a is greater than b;
507 //    -1 if b is greater than a.
508 //
509 //  Constraints is an array of enumerated types which
510 //  override mismatches.
511 //
512 int Compare(const Entryb &a, const Entryb &b,
513             char* Constraint, int* Terminus )
514 // Friend function that compares the keys in two entries.
515 // Returns -1 if a < b, 0 if a == b, and 1 if a > b.
516 {
517   if ( a.KeyCode == NULL ) {
518     throw ProgramError( "Entry Compare", "Search key format undefined" );
519   }
520   int i = 0;
521
522   char* Akey = (char*)a.key;
523   char* Bkey = (char*)b.key;
524
525   char* A_Terminus = Akey + a.Size() - a.BaseEntrySize();
526   while ( a.KeyCode[ i ] ) {
527     int rv = 0;
528     // Check to see if A is exhausted
529     // if so then we were looking for a leading edge
530     // key and we found it
531     if ( Akey == A_Terminus ) {
532       return rv;
533     }
534     switch ( a.KeyCode[i] ) {
535       case Entryb::VT_NULL:
536         return 0;
537       case Entryb::INT32: {
538           long lrv = *(long*) Akey - *(long*)Bkey;
539           if ( lrv < 0 ) rv = -1;
540           if ( lrv > 0 ) rv = 1;
```



```
541       }
542       Akey += sizeof( long ); Bkey += sizeof( long );
543       break;
544     case Entryb::UINT32:
545       if ( *(unsigned long*) Akey > *(unsigned long* ) Bkey  ) rv = 1;
546       if ( *(unsigned long*) Akey < *(unsigned long* ) Bkey  ) rv = -1;
547       Akey += sizeof( unsigned long); Bkey += sizeof( unsigned long );
548       break;
549     case Entryb::INT64: {
550         __int64 lrv = *(__int64*) Akey - *(__int64*)Bkey;
551         if ( lrv < 0 ) rv = -1;
552         if ( lrv > 0 ) rv = 1;
553       }
554       Akey += sizeof( __int64 );  Bkey += sizeof( __int64 );
555       break;
556
557     case Entryb::FLOAT_8:
558       if ( *( float *) Akey > *( float * ) Bkey )  rv = 1;
559       if ( *( float *) Akey < *( float * ) Bkey )  rv = -1;
560       Akey += sizeof( float ); Bkey += sizeof( float );
561       break;
562
563     case Entryb::FLOAT_10:
564       if ( *( long double *) Akey > *( long double * ) Bkey )  rv = 1;
565       if ( *( long double *) Akey < *( long double * ) Bkey )  rv = -1;
566       Akey += sizeof( long double ); Bkey += sizeof( long double );
567       break;
568     default:
569       {
570         short Alen = *(short*) Akey;  Akey += sizeof( short );
571         short Blen = *(short*) Bkey;  Bkey += sizeof( short );
572         rv =  CompVector( Akey, Bkey, min( Alen, Blen ) );
573         if ( rv == 0 ) {
574           if ( Alen < Blen )  rv = -1;
575           if ( Alen >  Blen ) rv = 1;
576         }
577         Akey += Alen; Bkey += Blen;
578         break;
579       }  // End Case Void
580     }    // End Switch
581
582   // Compare modifiers
583   if ( Constraint ) {
584     char WorkConstraint = Constraint[ i ];
585     // Not a match if NE is set
586     // if ( rv == 0 && WorkConstraint == Entryb::NE ) rv = 1;
587     switch ( WorkConstraint ) {
588         case Entryb::EQ:  break;
589         case Entryb::LT:  if ( rv == 0 ) rv = -1;
590         case Entryb::LE:  if ( rv == 1 ) rv = 0;
591         break;
592         case Entryb::GT:  if ( rv == 0) rv = 1;
593         case Entryb::GE:  if ( rv == -1) rv = 0;
594         break;
595         case Entryb::NE:  if ( rv == 0 ) { rv = 1; break; }
596                           return 0;
597         case Entryb::ALL: if ( rv ) rv = 0;
598           break;
599         default: break;
600     }  // switch( WorkConstraint )
```

```
601
602     ) // if Constraint
603     if ( rv == 0 ) { i++; continue; }  // Advance to next key
604
605     // If this parameter is provided, the failing key number is returned
606     if ( Terminus ) {
607       *Terminus = i;
608     } // if (Terminus)
609
610     if ( rv > 0 )  return 1;           // Less than
611     return -1;                         // Greater than
612   }  // End While
613
614   return 0;
615 }
616
617 int CompareExact(const Entryb &a, const Entryb &b)
618 // Friend function that compares not only the keys,
619 // but also the data fields as well.
620 // Returns -1 if a < b, 0 if a == b, and 1 if a > b.
621 {
622     int rv = Compare( a, b );
623     if ( rv ) return rv;
624     rv = memcmp( a.Data, b.Data, ENTRYDATASPACE );
625     if ( rv > 0 ) return 1;
626     if ( rv < 0 ) return -1;
627     return 0;
628 }
629
630 ////////////////////
631 // Vnode methods
632 ////////////////////
633
634 void Vnode::Collapse(int posn, int n)
635 // Collapse node data at byte position posn by n bytes
636 {
637   char *p = entries + posn;
638   memmove(p, p+n, tail-(posn+n));
639   tail -= n;
640 }
641
642 void Vnode::Expand(int posn, int n)
643 // Expand node data at byte position posn by n bytes
644 {
645   char *p = entries + posn;
646   memmove(p + n, p, tail-posn);
647   tail += n;
648 }
649
650 void Vnode::InsEntry(Entryb &e, int posn)
651 // Insert entry e into node at byte position posn
652 {
653   Expand(posn, e.Size());
654   memcpy( &EntryAt(posn), &e, e.Size() );
655   // EntryAt(posn) = e;
656 }
657
658 void Vnode::InsEntry(Entryb &e, int esize, int posn)
659 // Insert entry e into node at byte position posn
660 {
```



```
661   Expand(posn, esize);
662   memcpy( &EntryAt(posn), &e, e.Size() );
663   // EntryAt(posn) = e;
664 }
665
666 void Vnode::InsEntries(Vnode &b)
667 // Insert all entries of b into the front of this node
668 {
669   if (b.tail) {
670     memmove(entries+b.tail, entries, b.tail);
671     memmove(entries, b.entries, b.tail);
672   }
673 }
674
675 void Vnode::Split(Vnode &b, int split_posn)
676 // Move right half of data in this this node at split_posn
677 // into empty b.
678 {
679   b.tail = tail - split_posn; // Compute how much to move
680   memmove(b.entries, &EntryAt(split_posn), b.tail);
681   tail = split_posn;
682 }
683
684 void Vnode::Concatenate(Entryb &e, int esize)
685 // Concatenate entry e onto the end of this node
686 {
687   memmove(entries+tail, (char *)&e, esize);
688   tail += esize;
689 }
690
691 void Vnode::Concatenate(Vnode &b)
692 // Concatenate all entries from b onto the end of this node
693 {
694   if (b.tail) {
695     memmove(entries+tail, b.entries, b.tail);
696     tail += b.tail;
697   }
698 }
699
700 void Vnode::DelEntry(int posn)
701 // Delete the entry in node at byte position posn
702 {
703   Collapse(posn, EntryAt(posn).Size());
704 }
705
706 int Vnode::PrevPosn(int posn)
707 // Finds and returns the byte position of the entry just
708 // to the left of the one at posn. If no more entries to
709 // the left, it returns BEFORE_ENTRIES.
710 {
711   int tryposn, newposn;
712   tryposn = 0;
713   newposn = BEFORE_ENTRIES;
714   while (tryposn < posn) {
715     newposn = tryposn;
716     tryposn += EntryAt(tryposn).Size();
717   }
718   return newposn;
719 }
720
```

```
721 int Vnode::NextPosn(int posn, int stay_in_bounds)
722 // Returns the position of the next entry to the right of the
723 // entry at posn. If stay_in_bounds is 1, and we're at the
724 // last entry, we'll stay on last entry.
725 {
726    if (posn == BEFORE_ENTRIES) {
727       posn = 0;
728    }
729    else {
730       if (posn < tail || !stay_in_bounds)
731          posn += EntryAt(posn).Size();
732    }
733    return posn;
734 }
735
736
737 int Vnode::Search(const Entryb &e, int &posn)
738 // Tries to match e's key with the keys in the entries
739 // of this node. Passes back the posn of the entry
740 // giving a match, (if possible), or passes back
741 // the posn of the entry just before the entry that was
742 // too big. (This entry's right pointer thus points to
743 // where the search should continue.)
744 // Returns -1 if e's key is less than all keys in the node,
745 // returns 0 if there was an exact match, return 1 if e's
746 // key is greater than all keys in the node.
747 {
748    int p, nextposn, rv;
749
750    p = BEFORE_ENTRIES;
751    nextposn = 0;
752    rv = 1;
753    while (nextposn < tail) {
754       // While e's key > current entry's key
755       rv = Compare(e, EntryAt(nextposn));
756       if (rv <= 0) break;
757       p = nextposn;
758       nextposn += EntryAt(p).Size();
759    }
760    // Determine position based on whether exact match or not
761    posn = (rv == 0) ? nextposn : p;
762    return rv;
763 }
764
765 int Vnode::SearchExact(const Entryb &e, int &posn)
766 // Tries to match e's key with the keys in the entries
767 // of this node. Passes back the posn of the entry
768 // giving an exact match, (if possible), or passes back
769 // the posn of the entry just before the entry that was
770 // too big. (This entry's right pointer thus points to
771 // where the search should continue.)
772 // Returns -1 if e's key is less than all
773 // keys in the node, returns 0 if there was an exact
774 // match, return 1 if e's key is greater than all keys
775 // in the node.
776 {
777    int p, nextposn, rv;
778
779    p = BEFORE_ENTRIES;
780    nextposn = 0;
```



```
781    rv = 1;
782    while (nextposn < tail) {
783       // While e's key > current entry's key
784       rv = CompareExact(e, EntryAt(nextposn));
785       if (rv <= 0) break;
786       p = nextposn;
787       nextposn += EntryAt(p).Size();
788    }
789    // Determine position based on whether exact match or not
790    posn = (rv == 0) ? nextposn : p;
791    return rv;
792 }
793 Vnode::Vnode()
794 {
795    tail = 0; left = 0;
796 }
797
798 int Vnode::IsEmpty()
799 {
800    return tail == 0;
801 }
802
803 int Vnode::LastPosn()
804 {
805    return PrevPosn(tail);
806 }
807
808 int Vnode::AtTail()
809 // Returns the position after the last entry, (at the point
810 // where a new node could be concatenated.)
811 {
812    return tail;
813 }
814
815 int Vnode::TotalSize()
816 {
817    return tail;
818 }
819
820 int Vnode::InRange(int posn)
821 // Returns 1 if posn not before the before entry or
822 // after the last entry.
823 {
824    return posn >= 0 && posn < tail;
825 }
826
827 Entryb &Vnode::EntryAt(int posn)
828 // Returns the entry at byte position posn in the node
829 {
830    return *(Entryb *)(entries + posn);
831 }
832
833 long Vnode::Branch(int posn)
834 // Returns the branch (ie: pointer to child) corresponding
835 // to byte position posn in the node.
836 {
837    return (posn == BEFORE_ENTRIES) ? left : EntryAt(posn).right;
838 }
```

```
 1 ////////////////////////////////////////////////////////////////////////
 2 // vnode.h: Variable-order btree node class definition
 3 // Copyright(c) 1993 Azarone Software. All rights reserved.
 4 ////////////////////////////////////////////////////////////////////////
 5 #ifndef H_VNODE
 6 #define H_VNODE
 7 #include <stddef.h>
 8 #include <iostream.h>
 9
10 #if defined( WIN32 )
11 #define max(a,b)    (((a) > (b)) ? (a) : (b))
12 #define min(a,b)    (((a) < (b)) ? (a) : (b))
13 #endif
14
15 // Prototype for void and char area comparisons
16 struct Entryb;
17 int CompVector( void*, void*, int );
18 int Compare(const Entryb &a, const Entryb &b,
19                     char* Constraints = NULL, int* Terminus = NULL);
20 /*
21   Entry size constraints
22 */
23 #ifndef ENTRYKEYSPACE
24 #define ENTRYKEYSPACE   ( 256 )
25 #endif
26 #define ENTRYTRAILERSZ  ( sizeof( char* ) )
27
28 /*
29   Define the largest space for the data area of the
30   entry
31 */
32 #ifndef ENTRYDATASPACE
33 #define ENTRYDATASPACE ( 8 )
34 #endif
35
36 const int BEFORE_ENTRIES = -1;
37
38 // Forward reference to enable SelectVtree
39 class Vtree;
40
41 struct EXPORTSPEC Entryb {  // One entry of a node: base class version
42   long right;               // Points to right child
43   union {
44     long data_field;        // Data field, usually a record number or offset
45     char Data[ ENTRYDATASPACE]; // In case more space is needed for data
46     __int64 Int64;          // Save space for a big int
47   };
48   short EntrySize;          // How many actual bytes in this entry
49   char key[ENTRYKEYSPACE];  // Rest of key data to follow
50   //  Keycode is an array of characters containing enumerations
51   //  for each key segment.  Keycode is not is the "trailer" of the entry
52   //  and is not part of the index.
53   char* KeyCode; // SubKey Descriptor vector
54 public:
55   Entryb();
56   Entryb(const Entryb &src);
57   Entryb( char* KeyVector );
58   Entryb( Vtree& VT );
59
60   // Size of entry with no keys added
```



```
61   static short BaseEntrySize()
62     { return( sizeof( Entryb ) - ENTRYKEYSPACE - ENTRYTRAILERSZ ); }
63
64   // Entryb(const char *s, long df = 0, long r = 0);
65   // Extend key to include an element of this type
66   void BuildKey( char KeyCodeIx, void* KeyData, short DataLen = 0 );
67
68   void AddKeyElement( long  Value );
69   void AddKeyElement( long double  Value );
70   void AddKeyElement( unsigned long Value );
71   void AddKeyElement( short Value );
72   void AddKeyElement( char*  String );
73   void AddKeyElement( unsigned char* String );
74   void AddKeyElement( void* Ptr, short& DataLen );
75   void AddKeyElement( __int64 Value );
76
77   // Go to a particular element.
78   // ElemIx receives the number of the element found
79   void* SeekElement(  int ElementNo,
80                       char& ElementType,
81                       short& DataLen,
82                       short* ElemIx = NULL );
83   // Return the number of defined key items in the key
84   short GetElementCount();
85   void SetKeyCode( char* KC ) { KeyCode = KC; }
86   void SelectVtree( Vtree& VT );
87   void Clear();
88   // Retrieve pointer to composite key at position
89   void GetKeyValue( int KeyIx, void* DataArea, short& Cnt );
90
91   void operator=(const Entryb &src);
92   void       SetData( void*, int Len = sizeof(long) );
93   void       GetData( void* Targ, int Length = ENTRYDATASPACE ) {
94                memcpy( Targ, Data, Length );
95              }
96   unsigned   Size() const; // Returns size of actual data
97   void       printOn( ostream& os );
98   void       ConfirmKeyCode();
99   void       CheckOverrun();  // Check for overrun conditon on key
100
101  friend int Compare(const Entryb &a, const Entryb &b,
102                     char* Constraints, int* Terminus);
103  friend int CompareExact(const Entryb &a, const Entryb &b);
104
105  // duplicate the first part of a key
106  void DupKey( Entryb& Ent, int DupCnt );
107
108  enum {
109   VT_NULL,  // End identifier
110   INT32,    // Long integer     4 bytes
111   UINT32,   // Unsigned integer 4 bytes
112   INT64,    //
113   FLOAT_8,  // Floating integer 4 bytes
114   FLOAT_10, // Long double      10 bytes
115   CHAR,     // Character Array  variable
116   BINARY    // Array of binary data variable
117  };
118
119  enum {     // Constraint definitions
120    EQ = 1,  // Equal
```

```
    GE,     // Greater than or equal
    GT,     // Greater than
    LE,     // Less Than or equal
    LT,     // Less than
    NE,     // Not equal
    ALL     // Pass Everything
};
};
inline void Entryb::ConfirmKeyCode() {
    if ( KeyCode == NULL )
        throw ProgramError( "Entryb::AddKeyElement",
                            "Must set entry definition before building key" );
}
inline void Entryb::CheckOverrun() {
    if ( Size() - BaseEntrySize() > ENTRYKEYSPACE )
        throw ProgramError( "Entryb::AddKeyElement",
                            "Wrote Past End of Key Space" );
}

inline unsigned Entryb::Size() const
// Returns the total size of a entry, including
// the null byte terminator on the key
{
  return EntrySize;
}

struct Vnode {
  long left;        // Points to leftmost child
  int tell;         // Next insertion point
  char entries[1]; // Rest of entry bytes to follow
  Vnode();
  void Collapse(int posn, int n);
  void Expand(int posn, int n);
  void InsEntry(EntryB &e, int posn);
  void InsEntry(EntryB &e, int size, int posn);
  void InsEntries(Vnode &b);
  void DelEntry(int posn);
  void Split(Vnode &b, int split_posn);
  void Concatenate(EntryB &e, int size);
  void Concatenate(Vnode &b);
  int IsEmpty();
  int PrevPosn(int posn);
  int NextPosn(int posn, int stay_in_bounds = 1);
  int LastPosn();
  int AtTail();
  int TotalSize();
  int InRange(int posn);
  EntryB &EntryAt(int posn);
  long Branch(int posn);
  int Search(const EntryB &e, int &posn);
  int SearchExact(const EntryB &e, int &posn);
};

#endif
```

squilm C:\TMP\V1\VNODE.H Sat Sep 28 14:12:39 1996

```
 1 ////////////////////////////////////////////////////////////////////
 2 // file.cpp: File class methods.
 3 // Copyright (c) 1989-1993 Azarona Software. All rights reserved.
 4 ////////////////////////////////////////////////////////////////////
 5 #include <string.h>
 6 #include <stdlib.h>
 7 #include <errno.h>
 8 #include "vtfile2.h"
 9 #include <io.h>
10 #include <sys\stat.h>
11 #include <process.h>
12 #include "rela2.h"
13
14 // Allocate all necessary static data
15
16
17
18 ////////////////////////////////////////////////
19 // File methods
20 ////////////////////////////////////////////////
21
22 VtFile::VtFile()
23 // Creates a file manager object. Defaults to binary files.
24 {
25   status = GOOD; // good, read-only, and closed
26   refcnt = 0;
27   OwningRelation = NULL;
28 }
29
30 VtFile::~VtFile()
31 // Destructor for file object.
32 {
33   PrepareToDestroy();
34   OwningRelation = NULL;
35 }
36
37 //  This function calls the owning relation, if it exists
38 //  and causes that instance to flush all pending disk activity
39 void VtFile::FlushAllUsers(){
40   if ( OwningRelation ) {
41     OwningRelation->Flush();
42   }
43 }
44
45 //
46 //  This method tells the relation to shutdown all instances
47 //  of objects which use this file manager
48 void VtFile::ShutDownUsers() {
49   if ( OwningRelation ) {
50   // shutdown but save the instances for reload
51     OwningRelation->Unload( ON );
52   }
53 }
54 //
55 //  This method enables the instance to restart itself after
56 //  a rollback occurred.  The assumption is that the physical
57 //  file, and therefore the tree data was modified, possibly
58 //  meaning that cached data is out of date with the file
59 void VtFile::RestartUsers() {
60   if ( OwningRelation ) {
61     OwningRelation->Load();
62   }
63 //
64 //
65 //  This method attaches a relation to the current instance
66 //  to enable FlushAllUsers to tell the relation container
67 //  to flush all of it's indexes
68 //
69 void VtFile::AttachRelation( Relation* RP ) {
70   OwningRelation = RP;
71 }
72
73 int VtFile::__Create(char *fname)
74 // Creates and opens a new file named fname, with the previously
75 // set file_type. (Use SetFileType() before calling this to change
76 // the default which is stored in default_file_type). File
77 // truncated to zero if it already exists.
78 // Returns exception code. Throws some exceptions.
79 {
80   Close(); // First, close the current file if open.
81
82   status = GOOD | READ_WRITE;
83
84   // Create, truncate if already exists
85
86   SetName ( fname );
87   File_Mgr::Create ();  // May throw a file error
88   // if (GetFD () == 0) return CREATERR;
89   status |= OPEN;
90   /////lastop = FETCH; // So Store() works right the first time
91   lastop = SEEK;
92   return VT_SUCCESS;
93 }
94
95
96 int VtFile::Create(char *fname)
97 {
98   return __Create(fname);
99 }
100
101 int VtFile::Open(char *fname, AccessMode mode)
102 // Opens the existing file named fname, with the previously
103 // set file_type. (Use SetFileType() before calling this to change
104 // the default which is stored in default_file_type). If this
105 // object already pointing to an open file, that file is closed
106 // first. Returns exception code. May throw some exceptions.
107 {
108
109   Close(); // First, close the current file if open.
110
111   if (mode == READ_ONLY) {
112     status = GOOD; // good and read only
113   } else {
114     status = GOOD | READ_WRITE;
115   }
116   SetName ( fname );
117   File_Mgr::Open ();  // May throw a file error
118   // if (GetFD () == 0) return OPENERR;
119   status |= OPEN;
120   /////lastop = STORE; // So Fetch() works right the first time
```

```
121
122   lastop = SEEK;
123
124   return VT_SUCCESS;
125 }
126
127 void VtFile::PrepareToDestroy()
128 {
129   if (refcnt > 1) {
130      VtFileError VFE( "VtFile::Alloc", "Cannot release file in use" );
131      VFE.SetErrCode( VTFILE_DANGLING_PTR );
132      throw VFE;
133   }
134   Close();
135 }
136
137 void VtFile::Close(int flush)
138 // Closes the file if not already closed, flushing the
139 // basic header if flush = 1. Does nothing if in the
140 // error state. Checks for dangling references to
141 // this file. ALSO, sets file name to "closed file"
142 {
143   if (Ok()) {
144     if (flush && !ReadOnly()) Flush();
145 //     if ( File_Mgr::Close () != 0) FILE_THROW(CLOSERR);
146     File_Mgr::Close();
147   }
148   status &= ~OPEN; // Reset open bit
149 }
150
151 void VtFile::Flush()
152 // Flushes any internal buffers if needed.
153 {
154   if (ReadyForWriting()) {
155 //     if (fflush(fp) != 0) FILE_THROW(RWERR);
156     Seek (0L, CUR ); // So that my lastop logic works right
157   }
158 }
159
160 long VtFile::Seek(long ofs, VtFile::SeekDir sd)
161 // Moves the file pointer to the byte offset ofs,
162 // using the specified seek direction.
163 {
164   long rv = -1L;
165   //if (operator!()) FILE_THROW(NOTREADY);
166   if ( !Ok() ) {
167     VtFileError VFE( "VtFile::Seek", "File not ready" );
168     VFE.SetErrCode( VTFILE_NOT_READY );
169     throw VFE;
170   }
171   rv = File_Mgr::Seek ( ofs, (int)sd );
172   // if (rv == -1L) FILE_THROW(SEEKERR);
173   lastop = SEEK;
174 return rv;
175 }
176
177 unsigned VtFile::FetchSequential(void *d, unsigned nbytes, int exact)
178 // Fetches nbytes from the next sequential location in
179 // the file into buffer d. Returns number of bytes
180 // fetched, which might be zero if an error occurs.
```



```
181 // If exact == 1, then exactly nbytes must be read, otherwise
182 // an exception is thrown.
183 {
184   if ( !Ok() ) {
185     VtFileError VFE( "VtFile::FetchSequential", "File not ready" );
186     VFE.SetErrCode( VTFILE_NOT_READY );
187     throw VFE;
188   }
189   // if (operator!()) FILE_THROW(NOTREADY);
190
191   // Ensure seeks between intervening reads and writes,
192   // but optimize for sequential reads
193
194   if (lastop == STORE) {
195     Seek( 0L, CUR );
196     // if (rv == -1L) FILE_THROW(SEEKERR);
197     lastop = SEEK;
198   }
199
200 //  unsigned bytesmoved = ::fread(d, 1, nbytes, fp);
201
202   unsigned bytesmoved = Read( d, nbytes );
203
204   lastop = FETCH;
205
206   if (exact && bytesmoved != nbytes) {
207 //                FILE_THROW(eof(GetFD()) ? EOFERR : RWERR);
208     VtFileError VFE( "VtFile::FetchSequential", "Read Error" );
209     if ( eof( GetFD()) )
210       VFE.SetErrCode( VTFILE_EOF );
211     else
212       VFE.SetErrCode( VTFILE_READERR );
213     throw VFE;
214   }
215
216   return bytesmoved;
217 }
218
219
220 unsigned VtFile::Fetch(void *d, unsigned nbytes, long p, int exact)
221 // Fetches nbytes from address p into buffer d. The address
222 // is always interpreted to be from the beginning of the file.
223 // Returns number of bytes fetched, which may be zero if
224 // error occurs. If exact == 1, then exactly nbytes must be
225 // fetched, or an exception is thrown.
226 {
227   // if (operator!()) FILE_THROW(NOTREADY);
228   if ( !Ok() ) {
229     VtFileError VFE( "VtFile::Fetch", "File not ready" );
230     VFE.SetErrCode( VTFILE_NOT_READY );
231     throw VFE;
232   }
233   Seek( p, SEG );
234   // if (rv == -1L) FILE_THROW(SEEKERR);
235
236   lastop = SEEK;
237   unsigned bytesmoved = Read( d, nbytes );
238   lastop = FETCH;
239
240   if (exact && bytesmoved != nbytes) {
```

```
241 //    FILE_THROW(eof(GetFD()) ? EOFERR : RWERR);
242       VtFileError VFE( "VtFile::FetchSequential", "Read Error" );
243       if ( eof( GetFD()) )
244         VFE.SetErrCode( VTFILE_EOF  );
245       else
246         VFE.SetErrCode( VTFILE_READERR  );
247       throw VFE;
248     }
249     return bytesmoved;
250 }
251
252 void VtFile::StoreSequential( void *d, unsigned nbytes)
253 // Stores nbytes from buffer d to the next sequential
254 // location in the file. May throw an exception if not
255 // able to store nbytes of data.
256 {
257   if (!ReadyForWriting()) {
258     VtFileError VFE( "VtFile::StoreSequential", "File not ready for write" );
259     VFE.SetErrCode( VTFILE_NOT_READY  );
260     throw VFE;
261   }
262   // FILE_THROW(NOTWRITEABLE);
263
264   // Ensure seeks between intervening reads and writes,
265   // but optimize for sequential writes
266
267   if (lastop == FETCH) {
268     Seek( 0L, CUR );
269     // if (rv == -1L) FILE_THROW(SEEKERR);
270     lastop = SEEK;
271   }
272   Write( d, nbytes);
273   // if (bytesmoved != nbytes) FILE_THROW(eof(GetFD()) ? EOFERR : RWERR);
274
275   lastop = STORE;
276 }
277
278
279 void VtFile::Store( void *d, unsigned nbytes, long p)
280 // Stores nbytes from buffer d to addr p. May throw an
281 // exception if not able to store nbytes of data.
282 {
283   // FILE_THROW(NOTWRITEABLE);
284   if (!ReadyForWriting()) {
285     VtFileError VFE( "VtFile::Store", "File not set writeable" );
286     VFE.SetErrCode( VTFILE_NOT_READY  );
287     throw VFE;
288   }
289
290   Seek( p, BEG );
291   // if (rv == -1L) FILE_THROW(SEEKERR);
292   lastop = SEEK;
293   Write( d, nbytes);
294   // if (bytesmoved != nbytes) FILE_THROW(eof(GetFD()) ? EOFERR : RWERR);
295   lastop = STORE;
296 }
297
298
299 void VtFile::Rewind()
300 // Rewinds the file. All error bits reset as well.
```



```
301 {
302   if (!ReadyForReading())
303     VtFileError VFE( "VtFile::FetchSequential", "File not ready" );
304     VFE.SetErrCode( VTFILE_NOT_READY  );
305     throw VFE;
306   }
307   Seek( 0L, BEG );
308   // if (rv == -1L) FILE_THROW(SEEKERR);
309   ClearErr();
310   lastop = SEEK;
311 }
312
313
314 ////////////////////////////////////////////////////////////
315 // FilePtr Methods
316 ////////////////////////////////////////////////////////////
317
318 void FilePtr::Release()
319 {
320   if (objptr) {
321     objptr->refent--;
322     if (objptr->refent == 0) delete objptr;
323   }
324 }
325
326 void FilePtr::Bind(VtFile *p)
327 {
328   objptr = p;
329   if (objptr) objptr->refent++;
330 }
331
332 void FilePtr::NewBinding(VtFile *p)
333 // ASSUMES we'll be owning what p points to.
334 {
335   if (objptr != p) {
336     Release();
337     Bind(p);
338   }
339 }
340
341
342 FilePtr::FilePtr(VtFile *p)
343 // ASSUMES we'll be owning what p points to.
344 {
345   Bind(p);
346 }
347
348 FilePtr::FilePtr(const FilePtr &s)
349 {
350   Bind(s.objptr);
351 }
352
353
354 FilePtr::~FilePtr()
355 {
356   Release();
357 }
```

```
 1 ////////////////////////////////////////////////////////////
 2 // file.h: A file class to handle file-based objects.
 3 // Copyright(c) 1989-1993 Azarona Software. All rights reserved.
 4 ////////////////////////////////////////////////////////////
 5 #ifndef H_FILE
 6 #define H_FILE
 7 #include <stdio.h>
 8 #include "mgr_file.h"
 9
10 // Duplicate these defines here
11 #define VT_SUCCESS    ( 1 )
12 #define VT_FAIL       ( 0 )
13
14 // Error throw class for VtFile
15 // This one class handles all vtfile related errors at the
16 // low level
17
18 class VtFileError : public Exception {
19   char* WhyMsg;
20  public:
21   VtFileError( char* Function, char* why ) :
22     Exception( Function )
23   {
24       WhyMsg = why;
25       ExceptionClassCode = VTFILE_EXCEPTION_CODE;
26   }
27   virtual char* GetReportString() {
28     static char ReportString[ 256 ];
29     memset( ReportString, 0, 256 );
30     strcpy( ReportString, "VtFileError Thrown" );
31
32     if ( strlen( FunctionName ) ) {
33        strcpy( ReportString, "Function Name:  " );
34        strcat( ReportString, "  " );
35        strcat( ReportString, FunctionName );
36        strcat( ReportString, "\n" );
37      }
38
39      if ( strlen( WhyMsg ) ){
40       strcat( ReportString, WhyMsg );
41       strcat( ReportString, "\n" );
42     }
43     return ReportString;
44   }
45 };
46
47 // Numeric error codes
48 #define FMGR_HEADER_BOUNDS      ( 1 )
49 #define FMGR_BAD_FILE_HDR       ( 2 )
50 #define FMGR_BAD_FREEBLK_HDR    ( 3 )
51 #define FMGR_FILE_UNWRITEABLE   ( 4 )
52 #define VTFILE_DANGLING_PTR     ( 5 )
53 #define VTFILE_NOT_READY        ( 6 )
54 #define VTFILE_EOF              ( 7 )
55 #define VTFILE_READERR          ( 8 )
56 // #define FILE_THROW(c) Throw(c, __FILE__, __LINE__)
57
58 // Forward declaration for the Relation pointer
59 class Relation;
60
```

```
61 class VtFile  : public File_Mgr  {
62  public:
63
64
65   friend class FilePtr;
66   friend class FmgrPtr;
67
68   // File status bits look like this
69   //
70   // 3210
71   // |||+---- good = 1, bad = 0
72   // ||+----- open = 1, closed = 0
73   // |+------ read/write = 1, read/only = 0
74   // +------- text = 1, binary = 0
75   //
76   // WARNING: HAVEN'T HOOKED UP TEXT/BINARY BIT YET
77
78
79
80   enum FailStatus {
81     BAD = 0,
82     GOOD = 1
83   };
84
85   enum OpenStatus {
86     CLOSED = 0,
87     OPEN = 2
88   };
89
90   enum AccessMode {
91     READ_ONLY = 0,
92     READ_WRITE = 1
93   };
94
95
96   enum SeekDir  {
97     BEG=0, // Seek forward from beginning
98     CUR=1, // Seek forward from current
99     END=2  // Seek backward from end
100  };
101
102 protected:
103
104   enum IO_OP {
105     FETCH,
106     STORE,
107     SEEK
108   };
109
110   IO_OP lastop;        // Last I/O operation
111   int refcnt;          // Reference counter
112 //  Except exception;    // Code of last exception thrown
113   char status;         // See comments above
114
115 protected:
116
117   VtFile(const VtFile &) { }          // Disallow so refcnt stays ok
118   void operator=(const VtFile &) { } // Disallow so refcnt stays ok
119
120   virtual int __Create(char *fname); // Does a lot of dirty work
```

```
121   virtual void PrepareToDestroy();
122
123   // Provide the flush all users method to cause everyone
124   // to write their data to disk
125   void FlushAllUsers();
126   // Shutdown the instance before a rollback
127   void ShutDownUsers();
128   // Restart the instance after a rollback
129   void RestartUsers();
130
131  friend Relation;
132   void AttachRelation( Relation* RP );
133   Relation* OwningRelation;
134
135
136 public:
137   VtFile();
138   virtual ~VtFile();
139   int Create(char *fname); // Must *not* be virtual
140   virtual int Open(char *fname, AccessMode mode = READ_WRITE);
141   virtual void Close(int flush=1);
142   virtual void Flush();
143   unsigned FetchSequential(void *d, unsigned n, int exact = 1);
144   unsigned Fetch(void *d, unsigned n, long p, int exact = 1);
145   void StoreSequential( void *d, unsigned n);
146   void Store( void *d, unsigned n, long p);
147   long Seek(long ofs, SeekDir sd = BEG);
148   virtual void Rewind();
149   int IsOpen() const;
150   int ReadOnly() const;
151   int ReadyForWriting() const;
152   void ClearErr();
153   //virtual void ReportExcept(char *src, int ino);
154   //virtual void TakeExcept();
155   //virtual void Throw(Except ec, char *src=0, int ino=0);
156   int Ok() const;
157   //int operator!() const;
158   //operator const int () const;
159 };
160
161 inline int VtFile::IsOpen() const
162 // Returns true if file is open
163 {
164   return (status & OPEN);
165 }
166
167 inline int VtFile::ReadOnly() const
168 {
169   return (status & READ_WRITE) == 0;
170 }
171
172 inline void VtFile::ClearErr()
173 {
174 //  exception = SUCCESS;
175   status |= GOOD;
176 }
177
178 inline int VtFile::Ok() const
179 // We're okay if we're open and in a good state.
180 {
```



```
181   return (status & (GOOD | OPEN)) == (GOOD | OPEN);
182 }
183
184 inline int VtFile::ReadyForWriting() const
185 // We're ready for writing if we're ok and not read-only
186 {
187   return (status & (GOOD | OPEN | READ_WRITE)) ==
188          (GOOD | OPEN | READ_WRITE);
189 }
190 /*
191 inline int VtFile::operator!() const
192 // We're not okay if not open or not in a good state.
193 {
194   return (status & (GOOD | OPEN)) != (GOOD | OPEN);
195 }
196 */
197 /*
198 inline VtFile::operator const int() const
199 // Returns one if we're OK.
200 {
201   return (status & (GOOD | OPEN)) == (GOOD | OPEN);
202 }
203 */
204
205 //
206 // File Smart Pointer class
207 //
208
209 class FilePtr {
210 protected:
211   VtFile *objptr;
212   virtual void Release();
213   virtual void Bind(VtFile *p);
214   void NewBinding(VtFile *p);
215 public:
216   FilePtr(VtFile *p=0);
217   FilePtr(const FilePtr &a);
218   virtual ~FilePtr();
219   void operator=(const FilePtr &a);
220   void operator=(VtFile *p);
221   int operator!() const;
222   operator const void *() const;
223   VtFile &operator*() const;
224   VtFile *operator->() const;
225 };
226
227 inline void FilePtr::operator=(const FilePtr &a)
228 {
229   NewBinding(a.objptr);
230 }
231
232 inline void FilePtr::operator=(VtFile *p)
233 // ASSUMES we'll be owning what p points to.
234 {
235   NewBinding(p);
236 }
237
238 inline int FilePtr::operator!() const
239 {
240   return objptr == 0;
```

```
1
 1 ////////////////////////////////////////////////////////////
 2 // Vtree.cpp: Variable order methods.
 3 // Copyright(c) 1993 Azarona Software. All rights reserved.
 4 ////////////////////////////////////////////////////////////
 5 #include <string.h>
 6 #include "vtree.h"
 7
 8 // This has to be defined someplace
 9
10
11 Vtree::Vtree(int ns, int mks, int cs, int mh)
12 // Creates a variable-order btree with a node size of ns, a
13 // maximum key size of mks, a cache size of cs, and a maximum
14 // tree height of mh.
15 : f(0), path(mh), cache(cs, ns), root(cache)
16 {
17   // Key allocations should allow enough space for
18   // the base entry data.
19   max_key_size = mks;
20   memset( th.KeyVector, 0, KEYVECTORSZ );
21   // Ensure that nodesize is at least 3x larger than mks
22   if ( mks * 3 >= ns ) {
23     ProgramError VTE( "Vtree::Vtree", "Node size must be 3x key size" );
24     throw VTE;
25   }
26   th_Dirty = 0;
27 }
28
29 Vtree::~Vtree()
30 // Disconnects the vtree from its file and cache.
31 {
32   Disconnect();
33 }
34
35
36 ////////////////////////////////////////
37 // File management functions
38 ////////////////////////////////////////
39
40 int Vtree::Connect(FmgrPtr &fptr, Vtree::ConnectMode cm, long tha)
41 // Connect to an already open file. If cm == CREATE,
42 // we're creating a new tree (has nothing to do with whether
43 // we're creating the file itself.) Returns status code.
44 {
45   f = fptr;
46
47   cache.Connect(f);
48
49   if (cm == NEW_TREE) {
50
51     // May not have allocated tree header yet. If not,
52     // we allocate it, and by default store the address
53     // to it in the file's aux(0) static data area.
54
55     if (tha) {
56       th_addr = tha;
57     }
58     else {
59       th_addr = f->Alloc(sizeof(Header));
60       f->Aux(0) = th_addr;
61       f->WriteHdr(); // For safety
62
63
64     new(root) Vnode;
65     th.root_addr = root;
66     th.height = 1; th.num_entries = 0; th.num_nodes = 1;
67     th.node_size = cache.NodeSize();
68     th.max_key_size = max_key_size;
69
70     WriteHdr();
71     th_Dirty = 0;
72
73   }
74   else {
75
76     // If we haven't specified a tree header location,
77     // then ASSUME it's in the file's aux(0) static data area
78
79     if (tha) {
80       th_addr = tha;
81     }
82     else {
83   th_addr = f->Aux(0);
84     }
85
86     ReadHdr();
87     root = 0;    // Do these two statements to prevent dangling pointers
88     path.Clear();
89     if (th.node_size != cache.NodeSize()) {
90   cache.Reconfigure(th.node_size);
91     }
92     root = th.root_addr;
93
94   }
95
96   // Now, reset the tree's position to before first entry of root
97   Reset();
98
99   // Attach the keycode to the current vector
100  // EntryOb::KeyCode = th.KeyVector;
101  SetKeyVector();
102 #if defined( VTREE_DEBUG )
103    cout << "Loaded header: " << th_addr << endl;
104    cout << "Root Address   " << th.root_addr << endl;
105    cout << "Num Value:     " << th.num_entries << endl;
106 #endif
107
108  if (f->Ok()) return VT_SUCCESS;
109  else return VT_FAIL;
110 }
111
112 void Vtree::Disconnect()
113 // Disconnects the tree from the file and disconnects the
114 // cache from the file. If this tree owns the file, the file
115 // is closed. Otherwise, all we do is flush the buffers.
116 {
117
118 //C700: from  if (f) {
119   if (f!=0) {
120     if ( th_Dirty ) {
```

```
        WriteHdr();
        th_Dirty = 0;
      }
      root.Release();       // Do this so we won't have dangling ptrs
      path.Clear();         // Ditto
      cache.Disconnect();   // Disconnect cache from the file
      f = 0;                // And ditto with ourselves
    }
}

int Vtree::Create(char *fname)
// Create a new file to hold the tree. Disconnect from any file
// that we may be connected to first. Stores a pointer to the
// tree header in f->Aux(0).
// Returns status code of VT_SUCCESS OR VT_FAIL.
{
  Disconnect();
  FmgrPtr tmp(new Fmgr);
  if (tmp == 0) return VT_FAIL;
  tmp->Create(fname);
  if (!tmp->Ok()) return VT_FAIL;
  return Connect(tmp, NEW_TREE);
}
//
//  Set up the keycode vector in the tree header
//  Ensure that the Entryb vector is set
//
//  Either Create or Open must have been called
//  attaching and establishing this object in a file
//  since the key vector is written to the file's header
//
//  If KV is null this function resets the Entryb::Keycode to
//  point to this tree's key vector
//
int Vtree::SetKeyVector( char* KV ) {

  // Direct Entryb selection to this vector
  // Entryb::KeyCode = th.KeyVector;

  // Noop if KV is null
  if ( KV == NULL ) return VT_SUCCESS;
  // Cannot set key type after keys have been added
  if ( th.num_entries ) return VT_FAIL;
  // Only copy to the null character
  strcpy( th.KeyVector, KV );
  // Flush the tree's header
  WriteHdr();
  th_Dirty = 0;
  return VT_SUCCESS;
}


int Vtree::Open(char *fname, Fmgr::AccessMode mode, long tha)
// Opens an existing file which supposedly holds an existing tree.
// Disconnect from any file that we may be connected to first.
// Returns status code (VT_SUCCESS OR VT_FAIL).
{
  Disconnect();
  FmgrPtr tmp(new Fmgr);
  if (tmp == 0) return VT_FAIL;
  tmp->Open(fname, mode);
  if (!tmp->Ok()) return VT_FAIL;
  return Connect(tmp, EXISTING_TREE, tha);
}

void Vtree::WriteHdr()
{
  if (!f->ReadOnly())
     f->Store(&th, sizeof(Header), th_addr);
}

void Vtree::ReadHdr()
{
  f->Fetch(&th, sizeof(Header), th_addr);
}

void Vtree::Flush(int clear)
// Flushes all buffers (cache and headers) to disk
{
  if (f->Ok()) {
     // Want to check for dangling cptrs even if read-only
     cache.Flush(clear);
     if (f->ReadyForWriting()) {
        if ( th_Dirty ) {
          WriteHdr();
          th_Dirty = 0;
        }
        f->Flush();
     }
  }
}

//////////////////////////////////////////////////////////
// Low-level tree movement functions
//////////////////////////////////////////////////////////

void Vtree::Reset()
// Resets the tree by setting the current position to the
// left of the first entry in the root. That is, we're
// before any entries.
{
  root.posn = BEFORE_ENTRIES;
  path.Reset(root);
}

void Vtree::GoDown(long addr)
// Move down to the node at address addr. Presumably, the
// new node is a child of the current node.
{
  Footprint p(cache, addr);
  p.posn = BEFORE_ENTRIES;
  path.Curr().Release(); // Prevents dangling pointers
  if (!path.Push(p)) excp->VT_THROW(STKFULL);
}

void Vtree::GoUp()
// Move up to the current node's parent.
{
  path.Curr().Release(); // Prevents dangling pointers
  if (!path.Pop()) excp->VT_THROW(STKEMPTY);
```

```
241 }
242
243 int Vtree::PrevNodePosn(int posn)
244 // Move to entry before posn in current node. Returns
245 // offset of that entry. If already before the entries
246 // we remain there. New offset of position is returned.
247 {
248   posn = path.Curr()->PrevPosn(posn);
249   path.Curr().posn = posn;
250   return posn;
251 }
252
253 int Vtree::NextNodePosn(int posn)
254 // Move to entry after posn in current node. Returns
255 // offset of that entry. If at last entry, we stay there.
256 // New position's offset is returned.
257 {
258   posn = path.Curr()->NextPosn(posn);
259   path.Curr().posn = posn;
260   return posn;
261 }
262
263 int Vtree::ToLastNodePosn()
264 // Move to the last entry in the current node.
265 // New position's offset is returned.
266 {
267   int posn = path.Curr()->LastPosn();
268   path.Curr().posn = posn;
269   return posn;
270 }
271
272 ////////////////////////////////////////
273 // Mid-level tree movement routines.
274 ////////////////////////////////////////
275
276 void Vtree::AfterMaxEntry()
277 // Set position to just after the maximum entry in the tree.
278 {
279   int last_posn;
280   long p;
281
282   // Starting at root, go down, staying all the way
283   // to the right, till you reach leaf.
284   Reset();
285   while(1) {
286     last_posn = ToLastNodePosn();
287     if (last_posn >= 0) {
288       p = CurrEntryAt(last_posn).right;
289       if (p == 0) break;
290       GoDown(p);
291     }
292     else break;
293   }
294
295   // Position to just after last entry in the leaf node.
296   path.Curr().posn = path.Curr()->AtTail();
297 }
298
299
300
```

```
301 int Vtree::GoToSuccessor()
302 // Move to the successor of this node, which is always
303 // in a leaf. Returns EOT if there is no successor, else
304 // VT_SUCCESS.
305 {
306   long p;
307   int posn;
308
309   posn = path.Curr().posn;
310   if (posn != path.Curr()->AtTail()) {
311     p = path.Curr()->Branch(posn);
312     if (p == 0) return EOT;
313     // Now, scan down, staying always to the left of first
314     // entry in new node, until leaf is reached. Then, move
315     // to first entry of leaf.
316     while (p) {
317       GoDown(p);
318       p = path.Curr()->left;
319     }
320     NextNodePosn();
321     return VT_SUCCESS;
322   }
323   else return EOT;
324 }
325
326
327 int Vtree::Forward()
328 // Moves to the next entry in sorted order.
329 // Returns status code EOT (end-of-tree) or VT_SUCCESS.
330 {
331   long p;
332   int posn;
333
334   // If we're not after last entry in the current node,
335   // we need to go down to a leaf
336   posn = path.Curr().posn;
337   if (posn != path.Curr()->AtTail()) {
338     p = path.Curr()->Branch(posn);
339
340     // Now, scan down, staying always to the left of first
341     // entry in new node, until leaf is reached. Then, move
342     // to first entry of leaf.
343     while (p) {
344       GoDown(p);
345       p = path.Curr()->left;
346     }
347     NextNodePosn();
348   }
349
350
351
352
353
354
355   // We might be at the end of the current node (after the
356   // last entry.) If so, we must go up to parent, and go
357   // recursively up the tree.
358   while (path.Curr().posn == path.Curr()->AtTail()) {
359
360
```

C:\TMP\VTVTREE.CPP

```
361     // If we're at root, then there's no more
362     // entries left. Position to just after
363     // maximum entry, and return EOT.
364     if (path.Curr() == root) {
365        AfterMaxEntry();
366        return EOT;
367     }
368     // Else, move to parent node, and go to next entry
369     // of that node.
370     GoUp();
371     NextNodePosn(); // Note: if at tail, we stay at tail.
372   }
373
374   return VT_SUCCESS;
375 }
376
377 int Vtree::Backward()
378 // Moves to the previous entry in sorted order. Returns
379 // status code EOT (end-of-tree, actually in this case this
380 // means before beginning of tree) or VT_SUCCESS.
381 {
382   long p;
383
384   // Move back to previous entry of current node
385
386   PrevNodePosn();
387
388   // Walk down to leaf, staying to the right.
389
390   while(1) {
391     p = path.Curr()->Branch(path.Curr().posn);
392     if (p == 0) break;
393     GoDown(p);
394     // Go all the way to the last entry, (remember that node
395     // might be empty.)
396     ToLastNodePosn();
397   }
398
399   // If before first entry in node, (can happen if the original
400   // node was a leaf, especially when leaf is empty), then we
401   // must move up till we find a parent where we're not before
402   // first entry.
403
404   while(path.Curr().posn == BEFORE_ENTRIES) {
405     if (path.Curr() == root) {
406        // At root, so no more previous entries. Position
407        // to just before minimum entry, and return EOT.
408        BeforeMinEntry();
409        return EOT;
410     }
411     GoUp();
412   }
413
414   return VT_SUCCESS;
415 }
416
417
418 int Vtree::Forward(Entryb &e)
419 // Moves to the next entry in sorted order. If not at EOT,
420 // then e is loaded with a copy of the entry, otherwise
```



```
421 // e stays untouched.
422 // Returns status code EOT (end-of-tree) or VT_SUCCESS.
423 {
424   // Attach the keycode to the current vector
425   // Entryb::KeyCode = th.KeyVector;
426   // SetKeyVector();
427
428   int rv = Forward();
429   if (rv == VT_SUCCESS) e = CurrEntry();
430   return rv;
431 }
432
433 int Vtree::Backward(Entryb &e)
434 // Moves to the previous entry in sorted order. If not at EOT,
435 // then e is loaded with a copy of the entry, otherwise
436 // e stays untouched.
437 // Returns status code EOT (end-of-tree) or VT_SUCCESS.
438 {
439
440   // Attach the keycode to the current vector
441   // Entryb::KeyCode = th.KeyVector;
442   // SetKeyVector();
443
444   int rv = Backward();
445   if (rv == VT_SUCCESS) e = CurrEntry();
446   return rv;
447 }
448
449
450 ////////////////////////////////////////////
451 // Searching routines
452 ////////////////////////////////////////////
453
454 int Vtree::Search(const Entryb &e, int reset)
455 // Search the tree from the current position for the first
456 // matching entry. If reset == 1, then reset to the beginning
457 // of the tree before beginning the search.
458 // Returns status code (VT_SUCCESS or VT_FAIL).
459 {
460   int posn, found, rv;
461   long ptr;
462
463   if (reset) Reset();
464   found = VT_FAIL;
465
466   // Reset the keycode to the current tree
467   // Entryb::KeyCode = th.KeyVector;
468   SetKeyVector();
469
470   while (1) {
471     rv = path.Curr()->Search(e, posn);
472     path.Curr().posn = posn;      // Be sure to record position
473     if (rv == 0) return VT_SUCCESS; // Matching key found
474     ptr = path.Curr()->Branch(posn);
475     if (ptr == 0) break;
476     GoDown(ptr);
477   }
478
479   return found;
480 }
```

```
481
482 int Vtree::SearchExact(const Entryb &e, int reset)
483 // Search the tree from the current position on for an exact
484 // match. If reset == 1, then reset to the beginning of the tree
485 // before beginning the search.
486 // Returns VT_SUCCESS or VT_FAIL.
487 {
488   int posn;
489   long ptr;
490   int rv;
491
492   if (reset) Reset();
493
494   // Reset the keycode to the current tree
495   // Entryb::KeyCode = th.KeyVector;
496   // SetKeyVector();
497
498   while (1) {
499     rv = path.Curr()->SearchExact(e, posn);
500     path.Curr().posn = posn;  // Be sure to record position
501     if (rv == 0) return VT_SUCCESS;
502     ptr = path.Curr()->Branch(posn);
503     if (ptr == 0) break;
504     GoDown(ptr);
505   }
506
507   return VT_FAIL;
508 }
509 /*
510 int Vtree::FindFirst(Entryb &e)
511 // Finds first matching entry in tree by comparing keys. (The data
512 // field is not compared.) If found, loads in the rest of the entry,
513 // (ie: the data field.) If not found, e stays untouched.
514 // Returns status (VT_SUCCESS or VT_FAIL).
515 {
516   int rv = Search(e);
517   if (rv == VT_SUCCESS) e = CurrEntry();
518   return rv;
519 }
520 */
521
522 int Vtree::FindClosest(Entryb &e)
523 // Find the closest match (either key == key of e, or one with
524 // a key just larger than key of e, or if key of e would be past
525 // end of tree, the last entry of tree.) If success, e is loaded
526 // with key and data field. If no keys in tree, e stays untouched.
527 // Returns status (VT_SUCCESS or VT_FAIL if no entries in tree).
528 {
529   /*
530   int rv = Search(e);
531   if (rv == VT_SUCCESS) {
532     e = CurrEntry();
533   }
534   */
535   int rv = FindFirst( e );
536   if ( rv != VT_SUCCESS )
537   {  //GetCurr( e );
538     rv = Forward(e);
539     if (rv == EOT) rv = Backward(e);
540   }
```



```
541   if (rv == EOT) rv = VT_FAIL;
542   return rv;
543 }
544
545 //  The prior version did not work correctly with composite keys
546 //  The first encounter with a matching key caused a successful
547 //  return.  In this embodyment, the searched key must traverse
548 //  the tree backwards until a nonmatching key is found and then
549 //  advance one key unless the beginning of the tree is found
550
551 int Vtree::FindFirst(Entryb &e)
552 // Finds first matching entry in tree by comparing keys. (The data
553 // ffield is not compared.) If found, loads in the rest of the entry,
554 // (ie: the data ffield.) If not found, e stays untouched.
555 // Returns status (VT_SUCCESS or VT_FAIL).
556 {
557   int rv = Search(e);
558   if (rv == VT_SUCCESS) {
559     do {
560       // Walk backwards in the tree until a nonmatching key is found
561       // or we hit the beginning of the tree
562       rv = Backward( );
563       if ( rv != VT_SUCCESS ||
564            Compare( e, CurrEntry() ) )
565       {
566         // reload the subsequent key
567         Forward( e );
568         return VT_SUCCESS;
569       }
570     } while ( 1 );
571     // e = CurrEntry();
572   }
573   if (rv == EOT) rv = VT_FAIL;
574   return rv;
575 }
576
577
578 int Vtree::GetCurr(Entryb &e)
579 // Loads e with the current entry. If not an en
580 // entry, e isn't loaded, and VT_FAIL is returned,
581 // else VT_SUCCESS.
582 {
583   int posn = path.Curr().posn;
584   if (path.Curr()->InRange(posn)) {
585     e = CurrEntryAt(posn);
586     return VT_SUCCESS;
587   }
588   else return VT_FAIL;
589 }
590
591 ////////////////////////////////////////////////////////////////////////
592 // Insertion routines
593 ////////////////////////////////////////////////////////////////////////
594
595 int Vtree::Add(Entryb &entry, int reset)
596 // Add entry to tree. First, test to see if entry already there.
597 // If it is, report a duplicate entry error. If not, go ahead and
598 // add entry. If reset == 1, resets the path to the beginning of
599 // the tree after the add, otherwise, the path points to the
600 // entry just inserted.
```

```
601 // Returns VT_SUCCESS or DUPLENTRY.
602 {
603   // Check to see if the entry will fit
604   if ( (int)entry.Size() > th.max_key_size ) {
605     throw ProgramError( "Vtree::Add", "Entry is too large for tree" );
606   }
607
608   int rv = SearchExact(entry);
609   if (rv == VT_SUCCESS) return DUPLENTRY;
610   entry.right = 0;
611   rv = Insert(entry, reset);
612   if (rv == VT_SUCCESS) {
613     th.num_entries++;
614     th.Dirty = 1;
615 #if defined( VTREE_DEBUG )
616     cout << "Incrementing:  " << th.addr << endl;
617     cout << "Root Address   " << th.root_addr << endl;
618     cout << "New Value:     " << th.num_entries << endl;
619 #endif
620   }
621
622   return rv;
623 }
624
625 int Vtree::UpdateData( void* new_data, int WriteCnt )
626 // Updates the data field of the current entry. Returns
627 // DUPLENTRY if using the new data field would result in
628 // a duplicate entry, returns VT_FAIL if not currently on
629 // an entry, otherwise VT_SUCCESS.
630 {
631   if (path.Curr()->InRange(path.Curr().posn) == 0) return VT_FAIL;
632
633   // Test for possibly duplicate entry, get result in rv
634
635   // Entryb *e = new(max_key_size) Entryb(CurrEntry()); // <<<
636   Entryb *e = new Entryb( th.KeyVector);
637
638   // Load e with the current entry
639   GetCurr( *e );
640
641   char Old_Data[ ENTRYDATASPACE ];
642   // memset( Old_Data, 0, ENTRYDATASPACE );
643   memcpy( Old_Data, e->Data, ENTRYDATASPACE );
644
645   //long old_data = e->data_field;
646   // Copy in the new data to modify
647   memcpy( e->Data, new_data, WriteCnt );
648   //e->data_field = new_data;
649   int rv = SearchExact(*e);
650
651   // Now, reposition path to where it was before we started
652
653   //e->data_field = old_data;
654   memcpy( e->Data, Old_Data, ENTRYDATASPACE );
655
656   if (SearchExact(*e) != VT_SUCCESS) excp->VT_THROW(ASSERTERR);
657
658   // Notify of duplicate entry if determined
659   if (rv == VT_SUCCESS) {
660     rv = DUPLENTRY;
```



```
661   }
662   else {
663     // At this point, we are where we want to be. So update
664     // data field, inform cache.
665
666     // CurrEntry().data_field = new_data;
667     memcpy ( CurrEntry().Data, new_data, WriteCnt );
668     path.Curr()->SetDirty();
669     rv = VT_SUCCESS;
670   }
671   delete e;
672   return rv;
673 }
674
675 int Vtree::Insert(Entryb &entry, int reset)
676 // Inserts entry in the current node, after the current entry.
677 // May have to split the node and propagate splits up the tree.
678 // If reset == 1, resets the path to the beginning of the tree
679 // after the insert, otherwise, the path points to the entry
680 // just inserted.
681 // Returns status code.
682 {
683   Footprint rnode(cache); // Used when splitting
684   int split_occurred = 0, rv = VT_SUCCESS;
685
686   // Entryb *median_entry = new(max_key_size) Entryb;      // <<<
687   // Entryb *entry_to_insert = new(max_key_size) Entryb;  // <<<
688   Entryb *median_entry = new Entryb( th.KeyVector );
689   Entryb *entry_to_insert = new Entryb( th.KeyVector );
690
691   *entry_to_insert = entry;
692
693   // At this point, we're at a leaf, ready to insert entry
694   // at it's appropriate spot
695
696   while(1) { // Loop until we're done splitting and inserting
697
698     // Move to next position in current node, possibly
699     // past last entry of node. This will put us right
700     // where new entry can be inserted if there's room
701
702     int posn = path.Curr()->NextPosn(path.Curr().posn, 0);
703     path.Curr().posn = posn;
704
705     // May be able to insert entry w/o splitting node
706
707     if (path.Curr()->RoomFor(*entry_to_insert)) {
708       path.Curr()->InsEntry(*entry_to_insert, posn);
709       path.Curr()->SetDirty();
710       break;
711     }
712
713     else {
714
715       //
716       // We need to split. NOTE THAT WE MUST HAVE ROOM FOR
717       // AT LEAST TWO ENTRIES AT ALL TIMES FOR THIS TO WORK
718       //
719
720       split_occurred = 1;
```

```
721
722       // Create a new node to become the right node. Current
723       // node will be the left node.
724
725       new(rnode) Vnode; // Allocate a new node
726       if (rnode == 0) {
727          rv = ALLOCERR;
728          goto quit;
729       }
730       th.num_nodes++;
731       th_Dirty = 1;
732
733       // Move roughly half the nodes to the right node.
734
735       int splitting_root = (path.Curr() == root);
736       int curr_posn      = path.Curr().posn;
737       int split_posn     = path.Curr()->PrevPosn(path.Curr()->tail/2);
738
739       // Make correction in case we split the node wrong. At this
740       // point, our split position should keep at least one key to
741       // the left.
742
743       if (split_posn == 0)
744          split_posn = path.Curr()->NextPosn(0);
745
746       // However, if we will be inserting into the left node, then
747       // we'll take the median from the first entry from the right
748       // node. So ensure that the right node has one more entry.
749
750       if (curr_posn < split_posn)
751          split_posn = path.Curr()->PrevPosn(split_posn);
752
753       // At this point, it's possible for curr_posn == split_posn.
754       // This means the entry to be inserted is the median entry,
755       // UNLESS curr_posn == 0. In that case, the left node will
756       // get the new entry, and the median comes from the right node.
757       // THIS IS TRICKY FOLKS.
758
759       // We want to say *path.Curr()->Split(*rnode, split_posn) but
760       // BC++ 3.1 won't let us if we have inline functions turned on.
761       VnodeBucket &fixbc31 = *rnode;
762       path.Curr()->Split(fixbc31, split_posn);
763       path.Curr()->SetDirty();
764
765       // Determine which node, left or right, to insert entry into.
766
767       if (curr_posn > split_posn) {
768          // Put entry in right node. First entry of right
769          // node is the median. Extract it.
770          *median_entry = rnode->EntryAt(0);
771          rnode->DelEntry(0);
772          rnode->SetDirty();
773          // Insert entry into correct spot
774          curr_posn -= median_entry->Size();
775          rnode->InsEntry(*entry_to_insert, curr_posn - split_posn);
776          *entry_to_insert = *median_entry;
777       }
778       else if (curr_posn < split_posn || curr_posn == 0) {
779          // Entry goes into the left node. The alternative case
780          // curr_posn == 0 handles the possibility that we've split
```



```
781          // all entries into the right node, thus the left node
782          // is empty, and we must fill it with the new entry.
783          path.Curr()->InsEntry(*entry_to_insert, curr_posn);
784          path.Curr()->SetDirty();
785          *entry_to_insert = rnode->EntryAt(0);
786          rnode->DelEntry(0);
787          rnode->SetDirty();
788       }
789
790       // At this point, entry_to_insert is the entry to
791       // add to the parent. Adjust pointers accordingly.
792
793       rnode->left = entry_to_insert->right;
794       entry_to_insert->right = rnode;
795       rnode.Release();
796
797       if (splitting_root) {
798          // We've split the root, so grow new root
799          new(rnode) Vnode;
800          if (rnode == 0) {
801             rv = ALLOCERR;
802             goto quit;
803          }
804          th.root_addr = rnode;
805          th_Dirty = 1;
806          th.num_nodes++; th.height++;
807          rnode->left = root;
808          rnode->InsEntry(*entry_to_insert, 0);
809          root = rnode;
810          rnode.Release();
811          break; // Kick out of walk up tree
812       }
813       else {
814          // Fetch parent, insert entry there
815          GoUp();
816       }
817    } // if split
818 } // end of while
819
820 // If we had to split nodes, then reposition the
821 // path to the entry just inserted by searching
822 // for it from the beginning. Unless, we wish to
823 // reset the path.
824
825 if (reset) {
826    Reset();
827 }
828 else if (split_occurred) {
829    rv = SearchExact(entry);
830    if (rv != VT_SUCCESS) excp->VT_THROW(ASSERTERR); // Should find
831 }
832
833 quit:
834
835 delete entry_to_insert;
836 delete median_entry;
837
838 return rv;
839 }
840
```

```
841
842 ////////////////////////////////////////////////////////////////////
843 // Deletion routines
844 ////////////////////////////////////////////////////////////////////
845
846 int Vtree::Delete(Entryb &e, int reset)
847 // Delete entry e from the tree. An exact match, (both key and
848 // data pointer) must be found in tree or an error is returned.
849 // If reset == 1, it means to reset to beginning of tree after
850 // deletion, else it means to position the path to the entry
851 // just after the entry being deleted, if possible, or to the
852 // entry just before it, if there is one.
853 // Returns VT_SUCCESS/VT_FAIL status.
854 {
855   int rv = SearchExact(e); // Look for exact match
856   if (rv != VT_SUCCESS) return rv;
857   return DeleteCurr(reset);
858 }
859
860 int Vtree::DeleteCurr(int reset)
861 // Delete the current entry from the tree. We must actually
862 // be on an entry, or an error occurs.
863 // If reset == 1, it means to reset to beginning of tree after
864 // deletion, else it means to position the path to the entry
865 // just after the entry being deleted, if possible, or to the
866 // entry just before it, if there is one.
867 // Returns VT_SUCCESS/VT_FAIL status.
868 {
869   Footprint *save_pos;
870   int rv, have_successor, have_post_delete_entry;
871
872   if (path.Curr()->InRange(path.Curr().posn) == 0) return VT_FAIL;
873
874   // Entryb *post_delete_entry = new(max_key_size) Entryb; // <<<
875   Entryb *post_delete_entry = new Entryb( th.KeyVector);
876
877   // If not resetting after deletion, then find the next entry,
878   // which we'll position to after deleting. If no next entry,
879   // then try using the previous entry.
880
881   have_post_delete_entry = 0;
882
883   if (!reset) {
884     rv = Forward();
885     if (rv == VT_SUCCESS) {
886       have_post_delete_entry = 1;
887       *post_delete_entry = CurrEntry();
888       Backward(); // Position at entry to delete
889     }
890     else {
891       // We're one past the last of this point, and
892       // the entry to delete *is* the last, so try
893       // backing up two entries
894       Backward(); // Now we're on entry to delete
895       rv = Backward(); // Now we're one before
896       if (rv == VT_SUCCESS) {
897         have_post_delete_entry = 1;
898         *post_delete_entry = CurrEntry();
899       }
900       // else entry to delete is first entry, and we're before it
```



```
901       Forward(); // We're on entry to delete
902     }
903
904
905     // Save where we are, and find the successor
906
907     save_pos = path.Top();
908     have_successor = ToSuccessor() == VT_SUCCESS;
909
910     if (have_successor) {
911       // We have a successor. So delete current entry, insert
912       // successor in its place. Note that a split may occur.
913       *post_delete_entry = CurrEntry();
914       have_post_delete_entry = 1;
915       path.RewindTo(save_pos);
916       post_delete_entry->right = CurrEntry().right;
917       path.Curr()->DelEntry(path.Curr().posn);
918       path.Curr()->SetDirty();
919       PreviousPosn();
920       rv = Insert(*post_delete_entry);
921       if (rv != VT_SUCCESS) goto quit;
922       // Now, successor entry is actually in two places. Go to
923       // its original place in the leaf node so that we can
924       // delete that copy.
925       rv = ToSuccessor();
926       if (rv != VT_SUCCESS) goto quit;
927     }
928
929     // We're at a leaf at this point. Delete the current entry.
930
931     path.Curr()->DelEntry(path.Curr().posn);
932     path.Curr()->SetDirty();
933
934     // Keep tree balanced
935
936     rv = Balance();
937     if (rv != VT_SUCCESS) goto quit;
938
939     // Now, either reset to beginning of the tree, or go to the
940     // designated "post-delete" entry.
941
942     if (reset || (have_post_delete_entry) {
943       Reset();
944     }
945     else {
946       rv = SearchExact(*post_delete_entry);
947       if (rv != VT_SUCCESS) excp->VT_THROW(ASSERTERR); // Should find
948     }
949
950     rv = VT_SUCCESS;
951     th.num_entries--;
952     th_Dirty = 1;
953 #if defined( VTREE_DEBUG )
954       cout << "Decrementing:  " << th_addr << endl;
955       cout << "Root Address   " << th.root_addr << endl;
956       cout << "New Value:     " << th.num_entries << endl;
957 #endif
958
959 quit:
960
```

```
 961   delete post_delete_entry;
 962
 963   return rv;
 964 }
 965
 966
 967 int Vtree::Balance()
 968 // ASSUMES we just deleted an entry from the current node.
 969 // Next, we'll do rotations/merges/whatever's necessary to
 970 // balance the tree as much as possible. The balancing may
 971 // propagate up the tree.
 972 {
 973   Footprint parent(cache), curr(cache);
 974   Footprint lsibling(cache), rsibling(cache);
 975   Entryb *pentry, *lentry, *rentry, *rs_entry;
 976   int psize, lsize, rs_size, rsize, last_posn;
 977
 978   int rv;
 979
 980   // Entryb *temp_entry = new(max_key_size) Entryb; // <<<
 981   Entryb *temp_entry = new Entryb( th.KeyVector );
 982
 983   while(1) { // While not finished rotating/merging/etc.
 984
 985     curr = path.Curr();
 986     path.Curr().Release(); // Prevent dangling pointers
 987
 988     if (curr == root) {
 989        if (root->IsEmpty() && root->left) {
 990           // Empty root, we'll be shrinking tree
 991           long new_root_locn = root->left;
 992           root.Delete();
 993           th.num_nodes--; th.height--;
 994           th_Dirty = 1;
 995           root = new_root_locn;
 996           th.root_addr = new_root_locn;
 997        }
 998        break; // Nothing left to do
 999     }
1000
1001     // We know at this point that current node has a parent.
1002
1003     if (curr->IsSatisfied()) break; // No more balancing needed
1004
1005     parent = path.Parent(); // Guaranteed not null
1006
1007     if (parent.posn != BEFORE_ENTRIES) {
1008        pentry = &parent->EntryAt(parent.posn);
1009        psize = pentry->Size();
1010     }
1011     else {
1012        pentry = 0;
1013        psize = 0;
1014     }
1015
1016     int rposn = parent->NextPosn(parent.posn, 0);
1017
1018     if (rposn != parent->AtTail()) {
1019        // We have a right sibling
1020        rentry = &parent->EntryAt(rposn);
```



```
1021        rsize = rentry->Size();
1022        rsibling = rentry->right;
1023        rs_entry = &rsibling->EntryAt(0);
1024        rs_size = rs_entry->Size();
1025     }
1026     else {
1027        rentry = 0;
1028        rsize = 0;
1029        rsibling = 0;
1030        rs_entry = 0;
1031        rs_size = 0;
1032     }
1033
1034     if (parent->left != curr) {
1035        // We have a left sibling
1036        int posn = parent->PrevPosn(parent.posn);
1037        lsibling = parent->Branch(posn);
1038        last_posn = lsibling->LastPosn();
1039        lentry = &lsibling->EntryAt(last_posn);
1040        lsize = lentry->Size();
1041     }
1042     else {
1043        lsibling = 0;
1044        last_posn = 0;
1045        lentry = 0;
1046        lsize = 0;
1047     }
1048
1049     // Okay, now we know who all the players are.
1050     // See if a right rotation is possible and desirable.
1051     // If so, do it.
1052
1053     if (lsibling && lsibling->IsSatisfied()) {
1054        // Left sibling has plenty of entries. But would it
1055        // make any sense to take one from it?
1056        if (curr->IsEmpty() || lsibling->IsFat()) {
1057           // It would be desirable to do the rotation. But will
1058           // the lsibling's last entry fit into parent node, with
1059           // parent entry removed?
1060           if (parent->RoomFor(lsize - psize)) {
1061              // We're in luck. Proceed with right rotation.
1062              // Move entry from parent into right (current) node.
1063              // This entry will have the old left branch of the
1064              // current node.
1065              pentry->right = curr->left;
1066              curr->InsEntry(*pentry, psize, 0);
1067              parent->DelEntry(parent.posn);
1068              // Now, move rightmost entry of sibling node into parent.
1069              // This entry's branch becomes the right node's left pointer.
1070              curr->left = lentry->right;
1071              lentry->right = curr;
1072              parent->InsEntry(*lentry, lsize, parent.posn);
1073              lsibling->DelEntry(last_posn);
1074              // Tell cache that things have changed
1075              lsibling->SetDirty();
1076              parent->SetDirty();
1077              curr->SetDirty();
1078              goto uptree; // Examine parent node now
1079           }
1080     }
```

```
1081    }
1082
1083    // See if a left rotation is possible and desirable.
1084    // If so, do it.
1085
1086    if (rsibling && rsibling->IsSatisfied()) {
1087       // Right sibling has plenty of entries. But would it
1088       // make any sense to take one from it?
1089       if (curr->IsEmpty() || rsibling->IsFat()) {
1090          // It would be desirable to do the rotation. But will
1091          // the rsibling's last entry fit into parent node, with
1092          // parent entry removed?
1093          if (parent->RoomFor(rs_size - rsize)) {
1094             // We're in luck. Proceed with left rotation.
1095             // Move entry from parent into the current (left) node.
1096             // This entry gets the left pointer of the right node.
1097             rentry->right = rsibling->left;
1098             curr->Concatenate(*rentry, rsize);
1099             parent->DelEntry(rposn);
1100             // Now, move leftmost entry of right node into parent.
1101             // This entry will point to the right node. Also, we
1102             // have a new left branch of the right node.
1103             rsibling->left = rs_entry->right;
1104             rs_entry->right = rsibling;
1105             parent->InsEntry(*rs_entry, rs_size, rposn);
1106             rsibling->DelEntry(0);
1107             // Tell cache that things have changed
1108             rsibling->SetDirty();
1109             parent->SetDirty();
1110             curr->SetDirty();
1111             goto uptree; // Examine parent node now
1112          }
1113       }
1114    }
1115
1116    // See if we can merge the current node with its left sibling,
1117    // along with the parent entry.
1118
1119    if (lsibling && lsibling->RoomFor(psize + curr->TotalSize())) {
1120       pentry->right = curr->left;
1121       lsibling->Concatenate(*pentry, psize);
1122       // We want to say "lsibling->Concatenate(*curr)" but BC++ 3.1
1123       // won't let us if we have inline functions turned on.
1124       VnodeBucket &fixbc31 = *curr;
1125       lsibling->Concatenate(fixbc31);
1126       parent->DelEntry(parent.posn);
1127       curr.Delete();
1128       th.num_nodes--;
1129       th_dirty = 1;
1130       // Tell cache that things have changed
1131       lsibling->SetDirty();
1132       parent->SetDirty();
1133       goto uptree; // Examine parent node now
1134    }
1135
1136    // See if we can merge the current node with its right sibling,
1137    // along with the parent entry.
1138
1139    if (rsibling && curr->RoomFor(rsize + rsibling->TotalSize())) {
1140       rentry->right = rsibling->left;
```



```
1141       curr->Concatenate(*rentry, rsize);
1142       // We want to say "curr->Concatenate(*rsibling)" but BC++ 3.1
1143       // won't let us.
1144       VnodeBucket &fixbc31 = *rsibling;
1145       curr->Concatenate(fixbc31);
1146       parent->DelEntry(rposn);
1147       rsibling.Delete();
1148       th.num_nodes--;
1149       th_dirty = 1;
1150       // Tell cache that things have changed
1151       curr->SetDirty();
1152       parent->SetDirty();
1153       goto uptree; // Examine parent node now
1154    }
1155
1156    // At this point, we couldn't rotate or merge. However, the
1157    // current node may be empty, and we need to handle that.
1158    // We do this by forcing a rotation from left or right,
1159    // which is accomplished by splitting the parent.
1160
1161    if (curr->IsEmpty()) {
1162       if (lsibling) {
1163          // Do a forced right rotation. Note that lsibling
1164          // is guaranteed to have at least two entries.
1165          // Otherwise, we could have done a merge.
1166          pentry->right = curr->left;
1167          curr->InsEntry(*pentry, psize, 0);
1168          parent->DelEntry(parent.posn);
1169          // Prepare to put lentry into parent, and
1170          // then remove it from left sibling.
1171          curr->left = lentry->right;
1172          lentry->right = curr;
1173          *temp_entry = *lentry;
1174          lsibling->DelEntry(last_posn);
1175          lsibling->SetDirty();
1176          parent->SetDirty();
1177          curr->SetDirty();
1178          // Okay, insert temp_entry into parent, which
1179          // presumably will cause a split.
1180          GoUp();
1181          PrevNodePosn();
1182          rv = Insert(*temp_entry);
1183    if (rv != VT_SUCCESS) goto quit;
1184          break; // Done balancing
1185       }
1186       else if (rsibling) {
1187          // Do a forced left rotation. Note that rsibling
1188          // is guaranteed to have at least two entries.
1189          // Otherwise, we could have done a merge.
1190          rentry->right = rsibling->left;
1191          curr->Concatenate(*rentry, rsize);
1192          parent->DelEntry(rposn);
1193          rsibling->left = rs_entry->right;
1194          rs_entry->right = rsibling;
1195          *temp_entry = *rs_entry;
1196          rsibling->DelEntry(0);
1197          rsibling->SetDirty();
1198          parent->SetDirty();
1199          curr->SetDirty();
1200          // Okay, insert temp_entry into parent, which
```

```
1201            // presumably will cause a split.
1202            GoUp();
1203            PrevNodePosn();
1204            rv = Insert(*temp_entry);
1205        if (rv != VT_SUCCESS) goto quit;
1206            break; // Done balancing
1207          }
1208          else {
1209            // I don't believe we should get here
1210            excp->VT_THROW(ASSERTERR);
1211            break; // This probably won't get executed
1212          }
1213        }
1214
1215        break; // There's nothing we could or need to do, so quit
1216
1217 uptree:
1218
1219        parent.Release();      // Prevent dangling pointers
1220        rsibling.Release();
1221        lsibling.Release();
1222
1223        GoUp(); // Go up to parent, and try balancing.
1224
1225      } // End of while
1226
1227      rv = VT_SUCCESS;
1228
1229 quit:
1230
1231      delete temp_entry;
1232
1233      return rv;
1234 }
1235
1236
1237 void Vtree::PrintNode(Footprint &pb)
1238 {
1239      int i;
1240      long addr = pb;
1241
1242      cout << "addr:" << addr << "\t"
1243           << "tail:" << pb->tail << "\t"
1244           << "left:" << pb->left << endl;
1245
1246      // The printing an entry requires a valid keycode
1247      //  The referential keycodes have no such information
1248      EntryB TmpEntry( th.KeyVector );
1249
1250      if (pb->IsEmpty()) {
1251        cout << "EMPTY NODE" << endl;
1252        // printf("EMPTY NODE");
1253      }
1254      else {
1255        i = 0;
1256        while (i < pb->tail) {
1257          // pb->EntryAt(i).printOn(cout);
1258          TmpEntry = pb->EntryAt(i);
1259          TmpEntry.printOn( cout );
1260          i += pb->EntryAt(i).Size();
```



```
1261      }
1262 }
1263
1264
1265 /*
1266   PrintTree
1267   Print out the content of a vtree
1268
1269   Arguments:
1270
1271 */
1272 void Vtree::PrintTree(int i)
1273 {
1274   long p;
1275   int  end, i, j;
1276
1277   // Reset the keycode to the current tree
1278   // EntryB::KeyCode = th.KeyVector;
1279   SetKeyVector();
1280
1281   for (i=1; i<=i; i++) printf("  ");
1282   PrintNode(path.Curr());
1283   p = path.Curr()->left;
1284   if (p != 0) {
1285     GoDown(p);
1286     PrintTree(i+1);
1287     GoUp();
1288   }
1289   j = 0;
1290   end = path.Curr()->tail;
1291   while (j < end) {
1292     p = path.Curr()->EntryAt(j).right;
1293     if (p != 0) {
1294       GoDown(p);
1295       PrintTree(i+1);
1296       GoUp();
1297     }
1298     j += path.Curr()->EntryAt(j).Size();
1299   }
1300 }
1301
1302 void Vtree::Statistics( int full )
1303 {
1304   long avg_order = 0;
1305   if (th.num_nodes > 0)
1306     avg_order = th.num_entries / th.num_nodes + 1;
1307
1308   cout << "Tree Statistics" << endl;
1309   cout << "   Entries:   " << th.num_entries << endl
1310        << "   Nodes:     " << th.num_nodes   << endl
1311        << "   Height:    " << th.height      << endl
1312        << "   Avg Order: " << avg_order      << endl;
1313
1314
1315   //printf("Tree statistics: %u entries, %u nodes, %d levels, average order = %l
     d\n",
1316   //    th.num_entries, th.num_nodes, th.height, avg_order);
1317
1318   Report(cache, full);
1319 }
```

```
1  ////////////////////////////////////////////////////////////////////////////
2  // Vtree.h: Variable order btree class
3  // Copyright(c) 1993 Asarona Software. All rights reserved.
4  ////////////////////////////////////////////////////////////////////////////
5  #ifndef H_VTREE
6  #define H_VTREE
7
8  #include "fmgr.h"
9  #include "tropath.h"
10 #include <vnode.h>
11
12 #define VTREE_DEFAULT_NODESIZE  ( 512 )
13 #define VTREE_DEFAULT_MARKET    ( VTREE_DEFAULT_NODESIZE / 3 )
14 #define VTREE_DEFAULT_CACHESIZE ( 8 )
15 #define VTREE_DEFAULT_MAXHEIGHT ( 64 )  // Big tree for now
16 // Return status codes
17
18 //const int VT_SUCCESS    =  1;
19 // const int VT_FAIL       =  0;
20
21 //const int SUCCESS    =  1;
22 //const int FAIL       =  0;
23 const int EOT        = -1;
24 const int DUPLENTRY  = -2;
25 const int ALLOCERR   = -3;
26
27 //
28 //  Pad out the free header to a 512 byte boundary
29 //
30 #define KEYVECTORSZ ( 512 - 18 )
31 class EXPORTSPEC Vtree {  // Vtree file class
32 public:
33
34    struct Header {
35       long root_addr;
36       long num_nodes;
37       unsigned short node_size;
38       short max_key_size;
39       long num_entries;
40       short height;
41       char KeyVector[ KEYVECTORSZ ];
42    };
43
44    enum ConnectMode {
45       EXISTING_TREE,
46       NEW_TREE
47    };
48
49 protected:
50
51    FmgrPtr f;          // File the Vtree is connected to
52    long th_addr;       // Address of the Vtree header
53    friend Entryb;
54
55    Header th;          // Copy of Vtree header in memory
56    int th_dirty;       // Flag indicating if the tree header is dirty
57
58    TreePath path;      // Footprints thru the tree
59    VnodeCache cache;   // Node cache
60    Footprint root;     // Points to root of tree
61    int maxsize;        // Maximum size of any key, including null byte
62
63    void ReadHdr();     // Load in the Vtree header
64    void WriteHdr();    // Write out the Vtree header
65
66    int Search(const Entryb &e, int reset=1);
67    int SearchExact(const Entryb &e, int reset=1);
68    int Insert(Entryb &e, int reset = 0);
69
70    Entryb &CurrentEntryAt(int pos);
71
72    int PrevNodePoint(int pos);
73    int PrevNodePoint();
74    int NextNodePoint(int pos);
75    int NextNodePoint();
76    int IsLastNodePoint();
77
78    void GoBurn(long addr);
79    void GoUp();
80    int ToSuccessor();
81    int Balance();
82
83    // static void PrintNode(Footprint &pb);
84    void PrintNode(Footprint &pb);
85 public:
86
87    Vtree(int ns  = VTREE_DEFAULT_NODESIZE,
88          int ms  = VTREE_DEFAULT_MARKET,
89          int cs  = VTREE_DEFAULT_CACHESIZE,
90          int mh  = VTREE_DEFAULT_MAXHEIGHT );
91    ~Vtree();
92
93    void SetEntryKeyVector( Entryb& e ) { e.KeyCode = th.KeyVector; };
94
95    // File management routines
96
97    int Connect(FmgrPtr &fptr, Vtree::ConnectMode cm, long th_addr=0);
98    void Disconnect();
99    Entryb &CurrentEntry();
100
101   int Create(char *name);
102   int SetKeyVector( char* KV = NULL );  // Set the key pattern
103
104   int Open(char *fname, Fmgr::AccessMode mode, long th_addr=0);
105   void Flush(int clear=0);
106   void Close();
107
108   // Tree traversal routines
109
110   void Reset();
111   void BeforeMinEntry();
112   void AfterMaxEntry();
113   int Forward();
114   int Backward();
115   int Forward(Entryb &e);
116   int Backward(Entryb &e);
117
118   // Searching routines
119
120
```

3

```
  int FindFirst(Entryb &e);
  int FindExact(const Entryb &e);
  int FindClosest(Entryb &e);

  int GetCurr(Entryb &e);
  int Add(Entryb &e, int reset = 0);
  int UpdateData(void* new_data, int WriteCnt = ENTRYDATASPACE );
  int Delete(Entryb &e, int reset=0);
  int DeleteCurr(int reset=0);

  int IsEmpty() const;
  long NumEntries() const;

  int IsOpen() const;
  int Ok() const;
  //int operator!() const;
/****************
  operator int() const;
  void ClearErr();
*******************/

  void Statistics(int full);
  void PrintTree(int l);
  int  GetMaxKeySize() { return max_key_size; }
};

inline void Vtree::BeforeMinEntry()
// Same as doing a Reset()
{
  Reset();
}

inline Entryb &Vtree::CurrEntryAt(int posn)
{
  return path.Curr()->EntryAt(posn);
}

inline Entryb &Vtree::CurrEntry()
{
  return path.Curr()->EntryAt(path.Curr().posn);
}

inline int Vtree::PrevNodePosn()
{
  return PrevNodePosn(path.Curr().posn);
}

inline int Vtree::NextNodePosn()
{
  return NextNodePosn(path.Curr().posn);
}

inline void Vtree::Close()
// Disconnects the vtree from its file and cache.
{
  Disconnect();
}

inline int Vtree::IsEmpty() const
{
```

4

```
  return th.num_entries == 0;
}

inline long Vtree::NumEntries() const
{
  return th.num_entries;
}

inline int Vtree::IsOpen() const
{
  return f && f->IsOpen();
}

inline int Vtree::Ok() const
{
  return f && f->Ok();
}
/*
inline int Vtree::operator!() const
{
//  return f == 0 || f->operator!();
  return f == 0 || !(f->Ok());
}
*/
/***************************
inline Vtree::operator int() const
{
  return f && f->operator int();
}

inline void Vtree::ClearErr()
{
  if (f) f->ClearErr();
}
***************************/

inline int Vtree::FindExact(const Entryb &e)
// Looks for matching entry in tree by comparing keys and data
// fields. Returns SUCCESS or FAIL.
{
  return SearchExact(e);
}

#endif
```

000117

Drawings

Figure 1A-Basic System

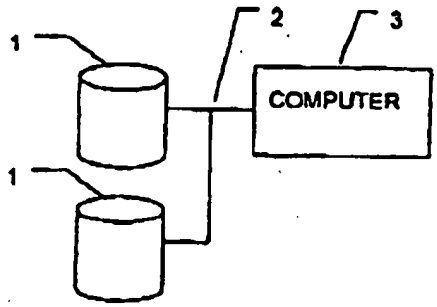


Figure 1B - Elemental Storage Units and Storage Address

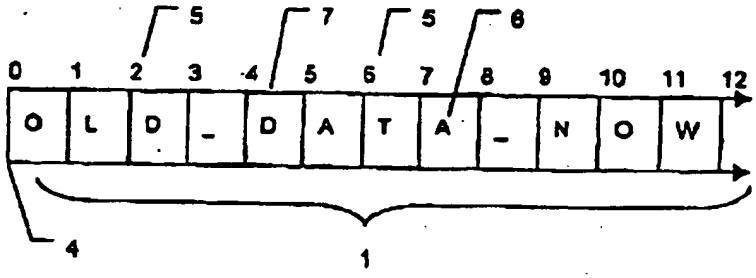


Figure 1c - Write Events to Computer Storage

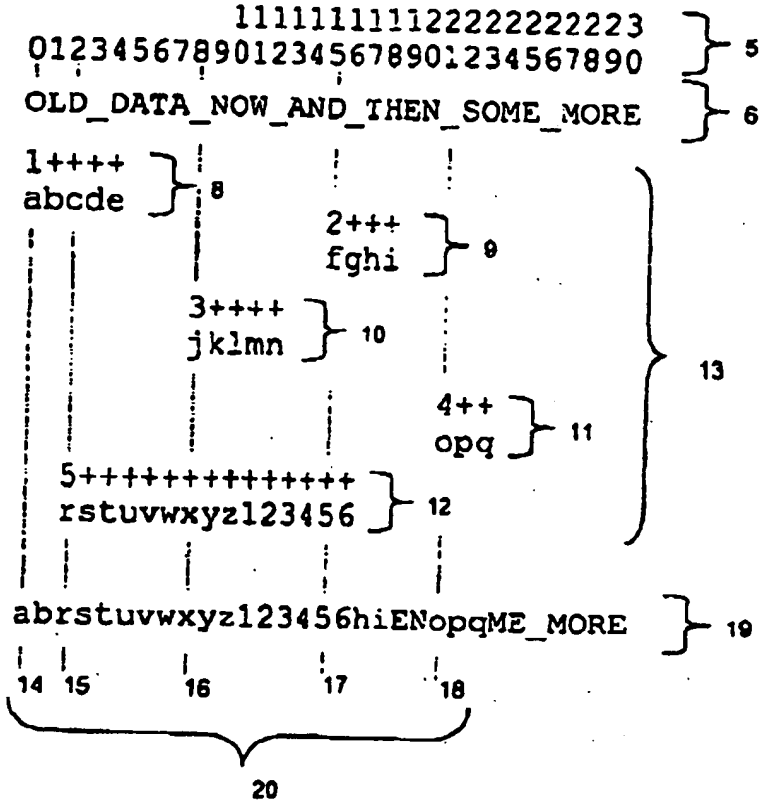


Figure 1D -Event Journal

Event Address (22)	Event Size (23)	Event Data (24)	Event Data Address (25)
0	5	abcde	16
15	4	fghi	37
8	5	jklmn	57
21	3	opq	78
2	15	rstuvwxyz123456	97

21

Figure 2A Event Map

Marker Origin Address (26)	Marker Event Span (27)	Marker Data Pointer (28)
0	2	16
2	15	97
17	2	39
21	3	78

28

Figure 2B – Creating the Event Map

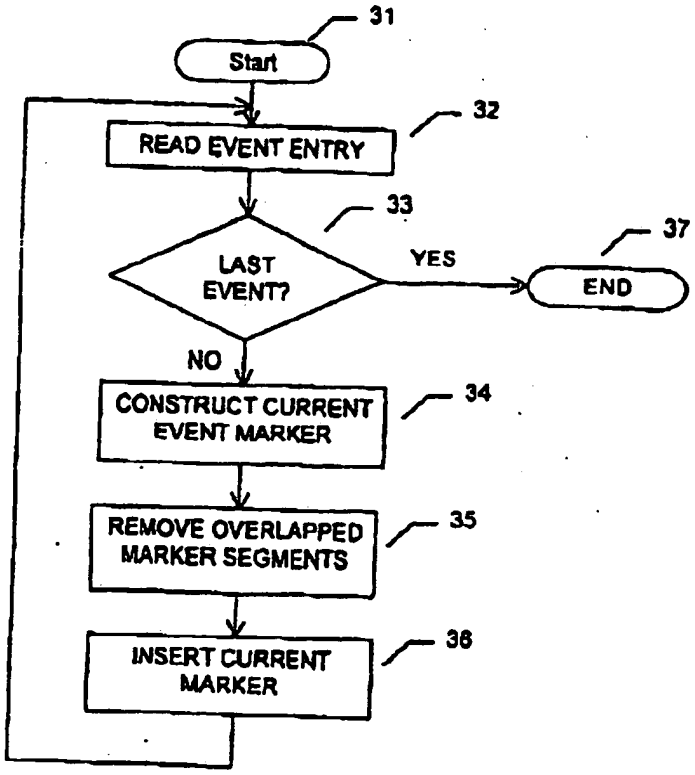


Figure 2C Construct Current Event Marker

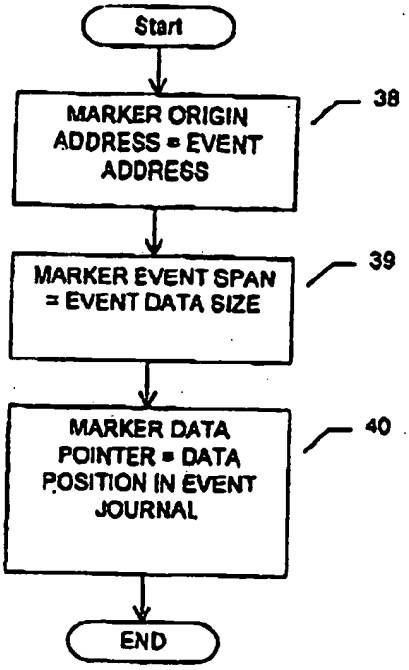


Figure 2D *– Remove Overlapped Marker Segments*

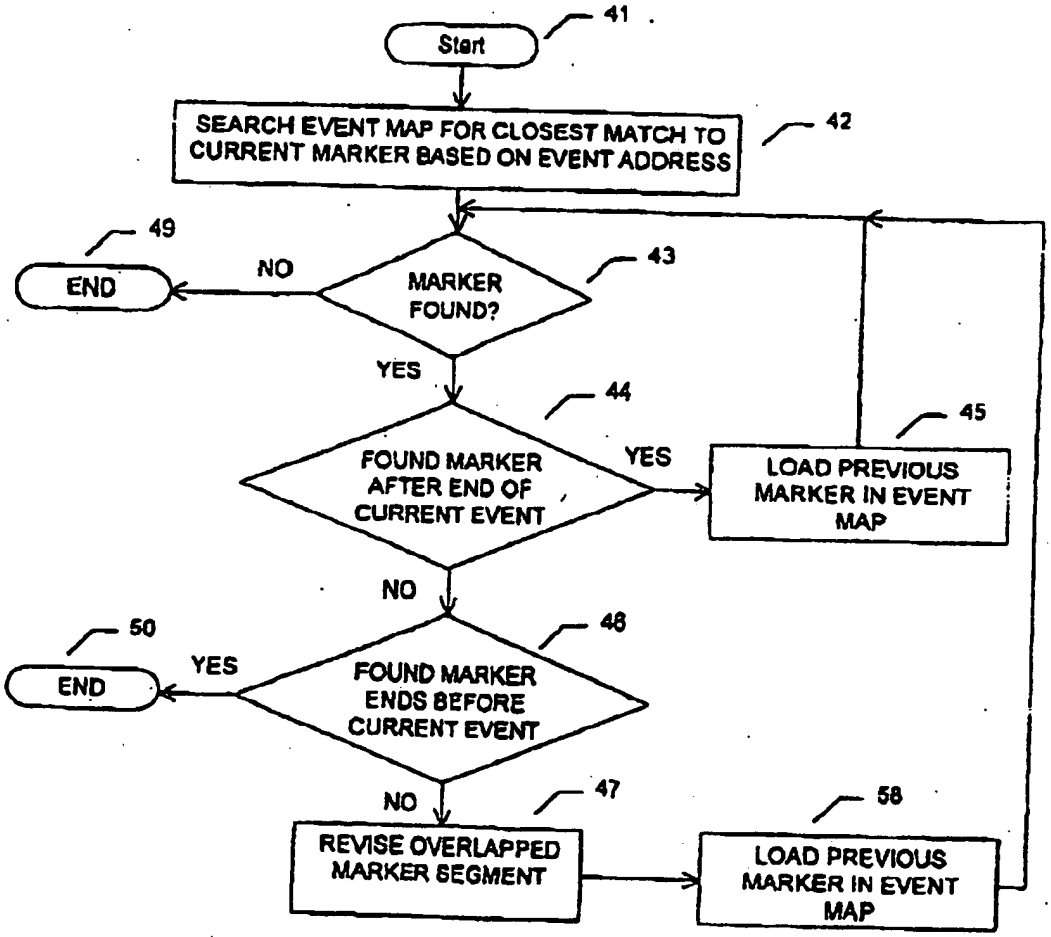


Figure 2B -Revise Overlapped Marker

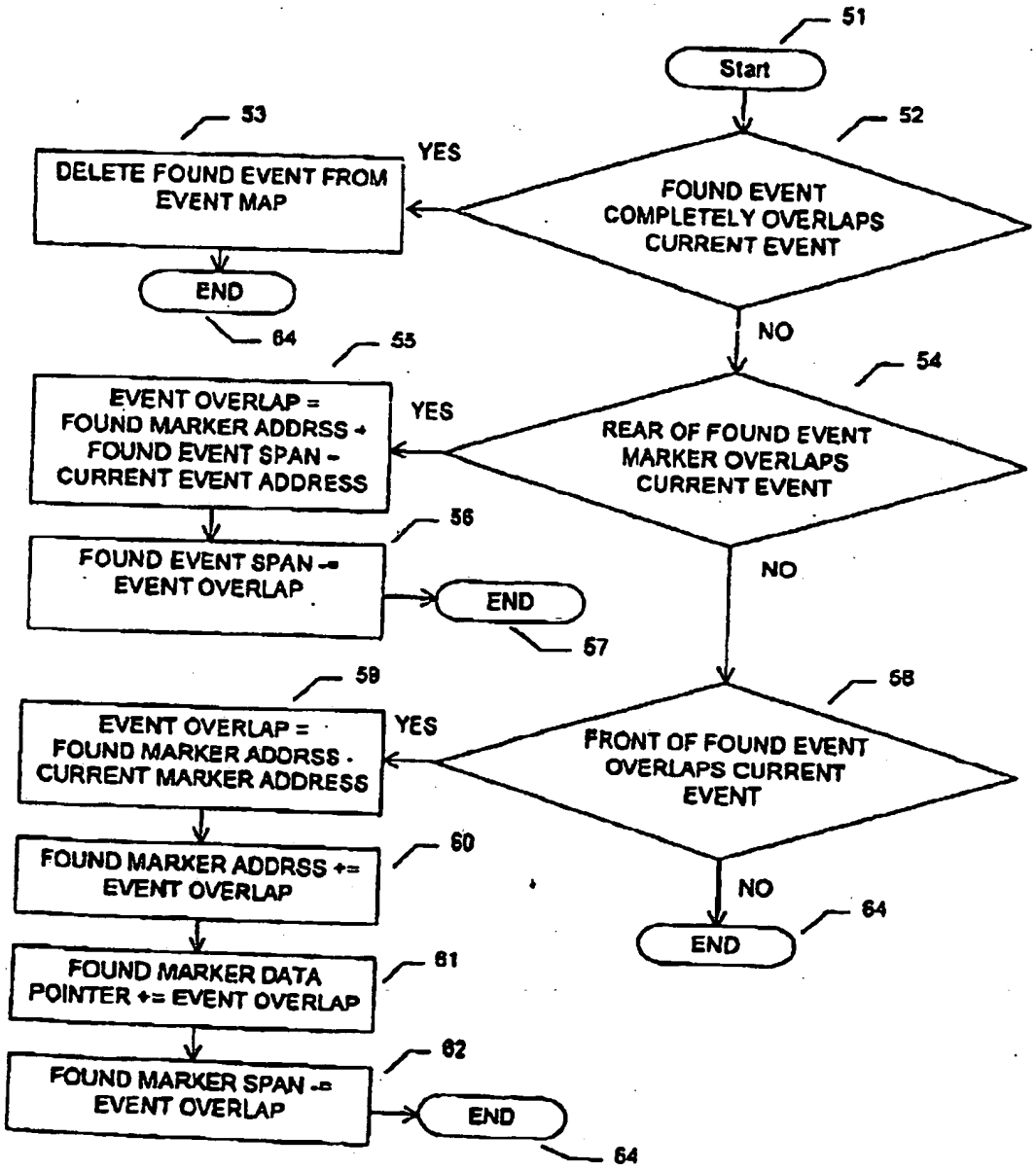


Figure 3 – Original and Updated Storage

```
                  11111111112222222223
012345678901234567890123456789O
OLD_DATA_NOW_AND_THEN_SOME_MORE
abrstuvwxyz123456hiENopqME_MORE
```

5
6
19

Figure 4A Components Fulfilling a Read Request

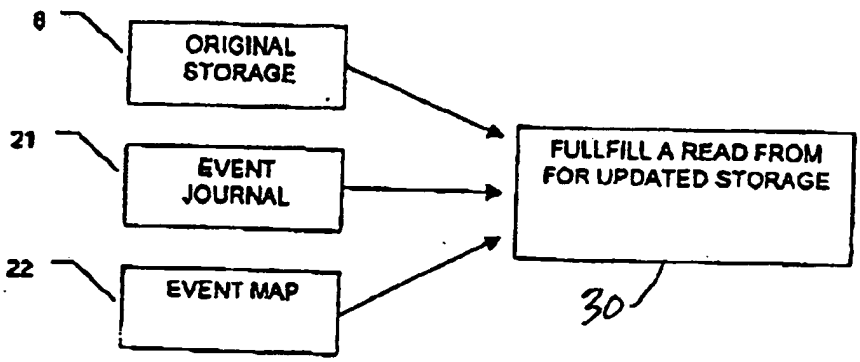


Figure 4B- Fulfilling a read request

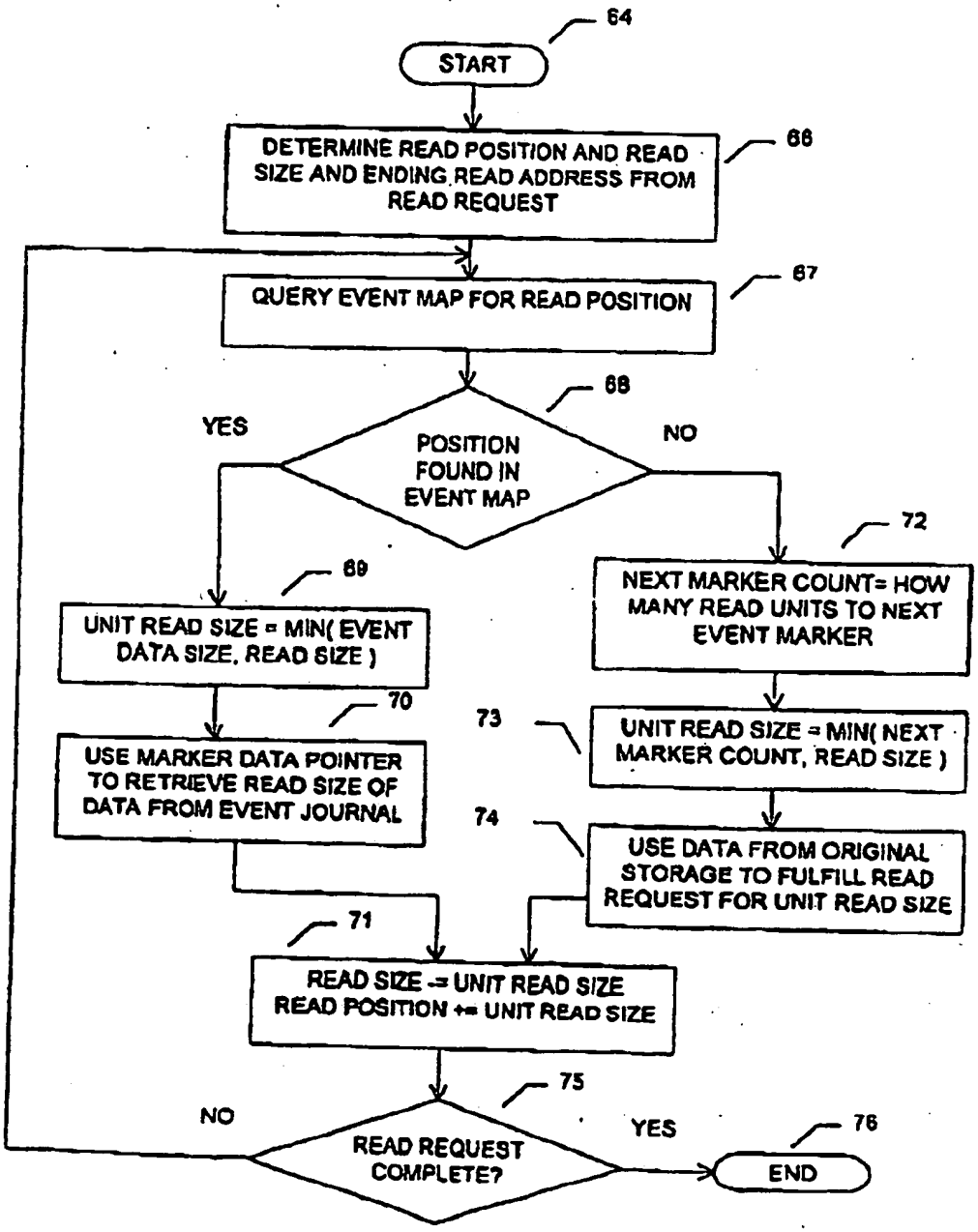


Figure 4C- Building a stream

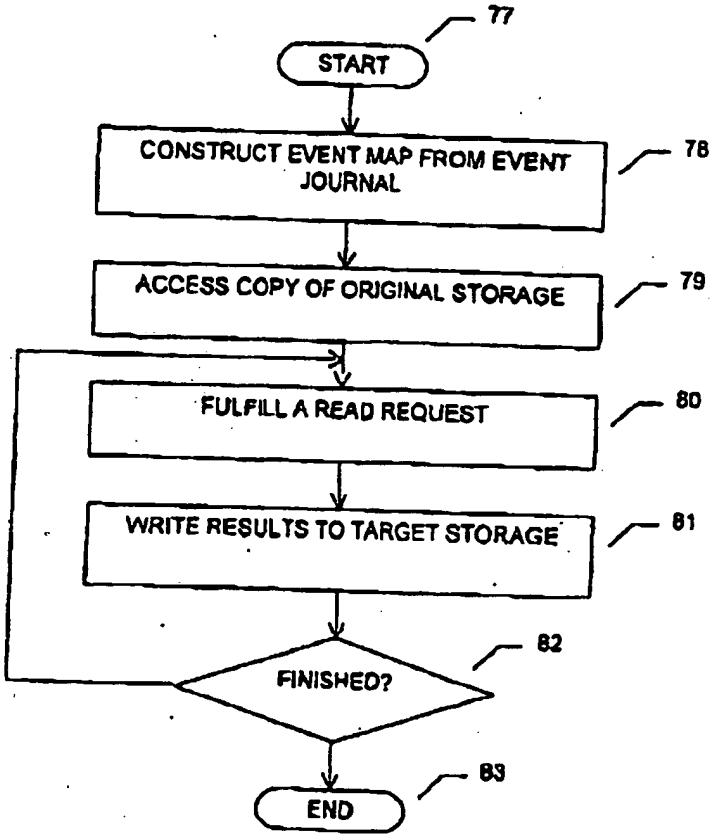


Figure 5 – Converting an Event Journal into a Delta

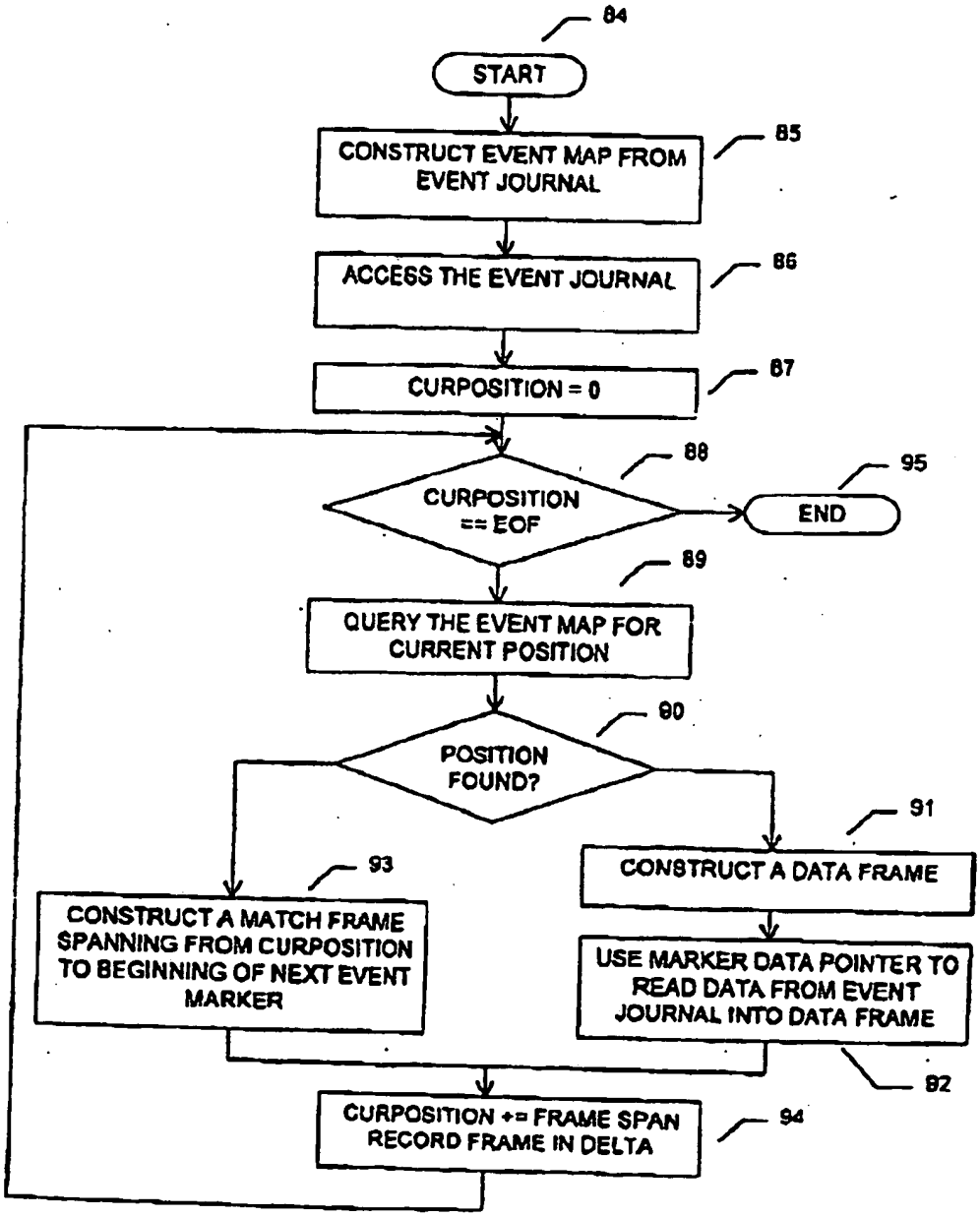


Figure 6A Reading and writing to a combination read-only storage and an event log

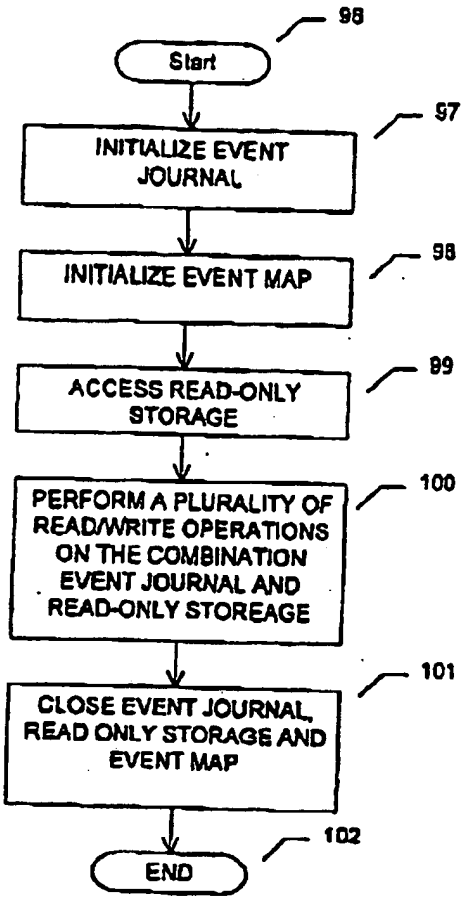


Figure 6B –Write to read-only storage event journal combination.

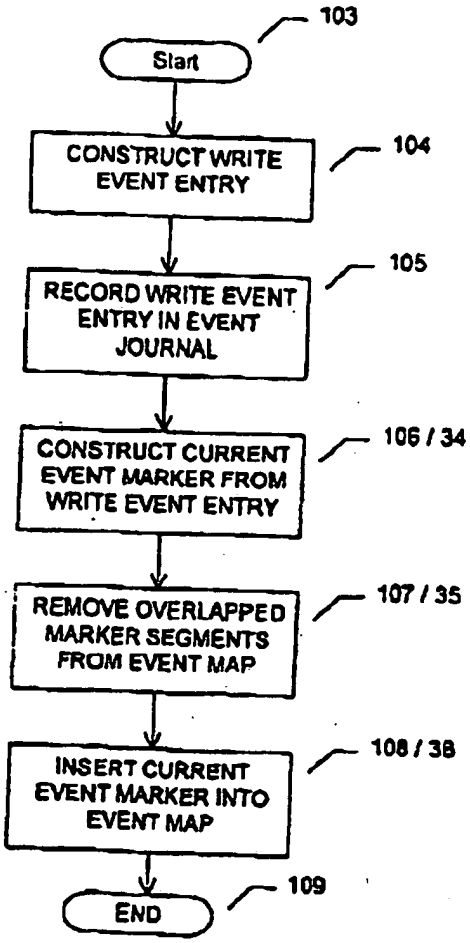


Figure 6C ~ Read from read-only storage event journal combination